图灵程序设计丛书

TCP Sockets编程

Working with TCP Sockets

[加] Jesse Storimer 著　　门佳 译

- 短小精悍的Sockets编程手册
- 代码示例详尽，供你举一反三
- 利用Ruby的语法糖实现高效构建

人民邮电出版社
POSTS & TELECOM PRESS

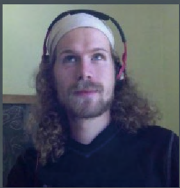
数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

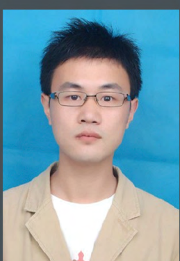
我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



Jesse Storimer

高级软件工程师与自出版作家，服务于美国电子商务方案提供商Shopify。痴迷于编程，编程之余喜欢阅读、园艺以及徒步旅行等户外运动。除了经常写技术博客文章外，他还著有《理解Unix进程》，*Working with Ruby Threads* 两本颇有影响力的电子书。



门佳

Unix / Linux shell、Perl、正则表达式爱好者。在2001年接触Linux后很快喜欢上该系统。对于Unix / Linux系统管理、Linux内核、Web技术研究颇多。工作之余，还喜欢探讨心理学，热衷出没于豆瓣和知乎。除了此书外，他的译作还包括《Linux Shell脚本攻略》和《理解Unix进程》等书。

图灵程序设计丛书

TCP Sockets编程

Working with TCP Sockets

[加] Jesse Storimer 著　门佳 译

人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（CIP）数据

TCP Sockets编程 /（加）斯托里默（Storimer,J.）著；门佳译. -- 北京：人民邮电出版社，2013.10
（图灵程序设计丛书）
书名原文：Working with TCP Sockets
ISBN 978-7-115-33052-9

Ⅰ. ①T… Ⅱ. ①斯… ②门… Ⅲ. ①计算机网络－程序设计 Ⅳ. ①TP393.09

中国版本图书馆CIP数据核字(2013)第215050号

内 容 提 要

本书通过循序渐进的方式，从最基础的概念到高级别的Ruby封装器，再到更复杂的应用，提供了开发成熟且功能强大的应用程序所必备的知识和技巧，帮助读者掌握在Ruby语言环境下，用套接字实现项目开发的任务和技术。

本书适合对TCP套接字感兴趣的读者阅读。

◆ 著 [加] Jesse Storimer
译 门 佳
责任编辑 丁晓昀
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：880×1230 1/32
印张：5.125
字数：125千字 2013年10月第1版
印数：1－3 500册 2013年10月北京第1次印刷
著作权合同登记号 图字：01-2013-3891号

定价：29.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223

反盗版热线：(010)67171154

广告经营许可证：京崇工商广字第0021号

版权声明

前　　言

套接字（socket）连接起了数字世界。

回想一下计算机的早年吧。那时候它是专供科学家使用的器物，他们用它来做数学运算以及模拟，那真是阳春白雪的东西啊。

计算机真正将普通人相互联系起来，已经是多年之后的事了。如今，计算机更多是由普罗大众在使用，科学家占的比例很小。人们能够随时随地同他人共享信息、互相交流，计算机就越发变得引人入胜。

正是网络编程——更确切地说，是一组特定的套接字编程API——的出现，才使得这一切成真。正在阅读本书的你可能每天都同他人进行在线联系，每天都在使用这些由计算机互联的想法所催生的技术。

所以说网络编程归根结底是关于共享与通信的。本书的目的在于使你更深入地理解网络编程的底层机制，为网络互联贡献出自己的一份力量。

我的故事

我仍记得与套接字初次相遇时的情景。嗯，那可实在算不上美好。

作为Web开发人员，我使用过各种HTTP API，也已经习惯了诸如REST、JSON这类高层概念。

后来，我不得不去集成一个域名注册API。

我拿到API文档就蒙了。文档中要求在某些私有主机名的随机端口上打开一个TCP套接字。这和Twitter API的工作方式一点都不一样！

文档不仅要求建立TCP套接字，而且并没有将数据编码为JSON，甚至连XML都不是。我必须使用它们自己的那一套基于行的协议（line protocol）。通过套接字发送专门格式化过的文本行，然后发送一个空行，接着是用于参数的一对键–值，最后跟上两个空行表明请求发送完毕。

随后还得用同样的方式读入响应的内容。当时我就在想："这搞的是哪出啊……"

我把这些东西拿给了我的同事看，他的反应和我一样。他也从来没用过这种API。他立刻提醒我道："我以前只在C语言中用过套接字。你可得小心点。在退出前一定得把套接字关闭，不然它就会一直处于打开状态。程序退出后，就很难将其关闭了。"

什么?！一直打开？协议？端口？我懵了。

后来另一位同事看了一眼，然后说："不是真的吧？你不知道怎么用套接字？要知道每次读取Web页面时，就是在使用套接字啊。你真应

该了解一下它的工作原理。”

我将此视为一次挑战。一下子理清这些概念实在有些吃不消，不过我也要全力以赴。尽管没少出错，但最终还是把它搞定了。就套接字而言，我对它的运用比以前更上了一层楼。对于工作中所依赖的那些技术也有了更深入的理解。这种感觉还真是不错。

借助于本书，我希望能够帮你减少一些我自己在初识套接字时所经历的烦恼，同时让你在深刻领悟这一系列技术后享受到神清气爽的感觉。

本书的读者对象

这本书的目标读者是工作在Unix或类Unix系统平台上的Ruby开发人员。

本书的读者应该熟悉Ruby，但不必掌握网络编程的相关概念，我会从网络编程的基础讲起。

本书所有实例代码使用Ruby 1.9编写，并没有在更早的版本上测试过。

本书的内容

本书包括三个主要部分。

第一部分介绍套接字编程的基础知识。你可以学到如何创建套接字，如何连接套接字以及如何共享数据。

第二部分阐述一些套接字编程的高级主题。在你学会了“Hello world”式的套接字编程之后，还需要掌握这些内容。

第三部分在真实场景中运用前两部分中学到的知识。这部分会教你如何在网络程序中使用并发技术。对于同一个问题，我们会采用多个架构模式解决，并对这些模式进行比较。

Berkeley 套接字 API

本书主要关注的是Berkeley套接字API及其用法。Berkeley套接字API最先于1983年出现在BSD操作系统4.2版本中。该操作系统是当时刚刚提出的TCP的首个实现。

Berkeley套接字API真正经受住了时间的检验。本书中所使用的API和被大多数现代编程语言所支持的API同1983年时的那套API如出一辙。

毫无疑问，Berkeley套接字API之所以能够屹立不倒的一个关键原因就是：*你可以在无需了解底层协议的情况下使用套接字*。这一点至关重要，我们会在后面详细探讨。

Berkeley套接字API是一种编程API，运作在实际的协议实现之上。它关注的是连接两个端点（endpoint）共享数据，而非处理分组和序列号。

已成业界标准的Berkeley套接字API是由C语言实现的，几乎所有用C编写的现代编程语言都会包含它的低层次接口。因此，我尽力使本书的大部分内容具有普适性。

也就是说，我并不是仅仅演示Ruby所提供的套接字API的包装类（wrapper class），而是先讲解低层次的API，然后再介绍Ruby的包装类，使你对套接字的理解不仅仅局限在Ruby中。

当使用其他的编程语言时，你仍然可以运用这里所学到的基础知识，利用低层次结构实现所需要的一切。

本书没有讲述的内容

之前我提到过Berkeley套接字API的优点就是你无需了解任何底层协议的细节。本书也会彻底贯彻这一点。

有些有关网络互联的书籍重点关注底层协议及其错综复杂的细节，甚至会在诸如UDP协议或是原始套接字上重新实现TCP。本书可不会讲这些内容。

我们采纳了Berkeley套接字API所奉行的观点，即使用者无需了解底层协议的实现。本书将重点放在如何使用API实现有用的功能，并尽可能关注如何完成实战任务。

不过有时候（例如从事性能优化），不够了解底层协议会使你无法正确地使用某种特性。这种情况下，我会解释一些必要的细节，以帮助你理解某些概念。

将话题再转回到协议上。我已经说过不会详细讨论TCP，同样也不会详细介绍其他诸如HTTP、FTP等应用层协议。在随后的章节中，我会在示例中用到一些协议，但并不会深入探讨它们。

如果你对协议感兴趣，我推荐Stevens的著作《TCP/IP详解》(*TCP/IP Illustrated*)[①]。

netcat

书中有很多地方都使用`netcat`工具创建了一些随意的连接,用来测试我们编写的各色程序。`netcat`(在终端下通常是`nc`)是一个Unix工具,可以用于创建TCP(以及UDP)连接并进行侦听。在使用套接字时,它可是你工具箱中必备的一件利器。

如果你的工作平台是Unix系统,那么`netcat`可能已经安装好了,运行本书的示例应该不会碰上什么问题。

致谢

首先我要感谢我的家人Sara和Inara。虽然她们并没有参与本书的编写,但是却以其独特的方式作出了贡献:给予我时间和空间从事写作,提醒我重要的事项。如果不是她们，此书断然不能成形。

接下来要感谢出色的审稿人。他们阅读了草稿，对每一页都提出了想法和意见,这一切都增进了本书的质量。非常感谢Jonathan Rudenberg、Henrik Nyh、Cody Fauser、Julien Boyer、Joshua Wehner、Mike Perham、Camilo Lopez、Pat Shaughnessy、Trevor Bramble、Ryan LeCompte、Joe James、Michael Bernstein、Jesus Castello以及Pradeepto Bhattacharya。

---

① http://www.amazon.com/TCP-Illustrated-Vol-Addison-Wesley-Professional/dp/0201633469.

目　　录

第1章

建立套接字

让我们结合例子开始套接字的学习之旅吧。

1.1 Ruby 的套接字库

Ruby的套接字类在默认情况下并不会被载入，它需要使用`require 'socket'`导入。其中包括了各种用于TCP套接字、UDP套接字的类，以及必要的基本类型。在本书中你会看到其中的部分内容。

> socket库是Ruby标准库的组成部分。同openssl、zlib及curses这些库类似，socket库与其所依赖的C语言库之间是thin binding关系，socket库在多个Ruby发布版中一直都很稳定。

在创建套接字之前不要忘记使用`require 'socket'`。

1.2 创建首个套接字

记住我之前的提醒，接下来让我们着手创建一个套接字。

```
                                 # ./code/snippets/create_socket.rb
require 'socket'

socket = Socket.new(Socket::AF_INET, Socket::SOCK_STREAM)
```

上面的代码在INET域创建了一个类型为STREAM的套接字。INET是internet的缩写，特别用于指代IPv4版本的套接字。

STREAM表示将使用数据流进行通信，该功能由TCP提供。如果你指定的是DGRAM（datagram的缩写，数据报），则表示UDP套接字。套接字的类型用来告诉内核需要创建什么样的套接字。

1.3 什么是端点

在谈到IPv4的时候，我提到了几个新名词。继续新的内容之前，让我们先学习一下IPv4和寻址。

在两个套接字之间进行通信，就需要知道如何找到对方。这很像是打电话：如果你想和某人进行电话交流，必须知道对方的电话号码。

套接字使用IP地址将消息指向特定的主机。主机由唯一的IP地址来标识，IP地址就是主机的“电话号码”。

上面我特别提到了IPv4地址。IPv4地址通常看起来像这样：192.168.0.1。它是由点号相连接的4个小于等于255的数字。这东西能做什么？配置了IP地址的主机可以向另一台同样配置了IP地址的主机发送数据。

IP地址电话簿

你知道想要与之对话的主机的地址后，就很容易想象套接字通信了，但是怎样才能获取到那个地址呢？需要把它背下来吗，还是写在纸上？谢天谢地，都不需要。

你之前可能听说过DNS。它是一个用来将主机名映射到IP地址的系统。有了它，你就无需记忆主机的地址，不过得记住它的名字。随后可以让DNS将主机名解析成地址。即便是地址发生了变动，主机名总是能够将你引向正确的位置。真棒！

1.4 环回地址

IP地址未必总是指向远端主机。尤其是在研发阶段，你通常需要连接自己本地主机上的套接字。

多数系统都定义了环回接口（loopback interface）。和网卡接口不同，这是一个和硬件无关、完全虚拟的接口。发送到环回接口上的数据立即会在同一个接口上被接收。配合环回地址，你就可以将网络搭建在本地主机中。

环回接口对应的主机名是`localhost`，对应的IP地址通常是`127.0.0.1`。这些都定义在系统的hosts文件中。

1.5 IPv6

我说起过几次IPv4，但是并没有提过IPv6。IPv6是另一种IP地址寻址方案。

干嘛要有两种寻址方案？因为IPv4地址已经用完了。[①]IPv4由4组数字组成，各自的范围在0～255。每一组数字可以用8位二进制数字来表示，合计共需32位二进制。这意味着有2^{32}或43亿个地址。这是个不小的数字，不过想象一下你每天看到有多少接入网络的设备……就不会惊讶于IP地址的枯竭了。

今天IPv6地址空间看起来非常大，不过随着IPv4地址逐渐耗尽，IPv6一定会越来越重要。IPv6采用了一种不同的格式，可以拥有天文数字级别的独立IP地址。

不过大部分时间你无需手动输入这些地址，无论使用哪种寻址方案，结果都一样。

1.6　端口

对于端点而言，还有另外一个重要的方面——端口号。继续我们那个电话的例子：如果你要和办公楼中的某人进行通话，就得拨通他们的电话号码，然后再拨分机号。端口号就是套接字端点的“分机号”。

对于每个套接字而言，IP地址和端口号的组合必须是唯一的。所以在同一个侦听端口上可以有两个套接字，一个使用IPv4地址，另一个使用IPv6地址，但是这两个套接字不能都使用同一个IPv4地址。

① http://www.nro.net/news/ipv4-free-pool-depleted.

若没有端口号，一台主机一次只能够支持一个套接字。将每个活动套接字与特定的端口号结合起来，主机便可以同时支持上千个套接字。

> **我该使用哪个端口号？**
>
> DNS没法解决这个问题，不过我们可以借助已明确定义的端口号列表。
>
> 例如，HTTP默认在端口80上进行通信，FTP的端口是21。实际上有一个组织负责维护这个列表。①在下一章会用到更多的端口号。

1.7　创建第二个套接字

现在让我们来尝点Ruby提供的语法糖（syntactic sugar）。

尽管就创建套接字来说，有很多更高级别的抽象，但Ruby可以让你使用符号（而非常量）来描述各种选项。因此你可以用`:INET`和`:STREAM`分别描述`Socket::AF_INET`以及`Socket::SOCK_ STREAM`。下面是一个在IPv6域中创建TCP套接字的示例：

```
                    # ./code/snippets/create_socket_memoized.rb
require 'socket'

socket = Socket.new(:INET6, :STREAM)
```

这段代码创建了一个套接字，不过还不能同其他套接字交换数据。下一章我们会看到如何使用类似的套接字完成实际的工作。

---

① http://www.iana.org/.

1.8　文档

现在该讲到文档了。从事套接字编程的一件妙事就是系统中已经包含了大量有帮助的文档。可以查找文档的地方主要有两处：(1) 手册页；(2) ri。

下面是简单说明。

(1) **Unix手册页**提供了有关底层系统函数（C语言代码）的文档。Ruby的套接 字库便是在此基础上构建的。尽管手册页涉及的都是底层内容，但是可以让你了解某个系统调用的作用，这一点正是Ruby文档所欠缺的。它还可以告诉你该系统调用可能出现的错误代码。

例如在上面的代码中，我们使用了`Socket.new`。它会映射到系统函数`socket()`，该函数负责创建一个套接字。可以使用下面这个命令来查看它的用法：

```
$ man 2 socket
```

注意到2没？这告诉man程序查看手册页的第2节。手册页被划分成若干节。

- 节1：一般命令（shell程序）。
- 节2：系统调用。
- 节3：C库函数。
- 节4：特殊文件。
- 节5：文件格式。
- 节7：提供了各种话题的综述。tcp(7)就很有意思。

我会使用这样的语法引用手册页：socket(2)。它引用的是socket手册页的第2节。这样的语法是很有必要的，因为一些手册页存在于多个节之中，例如stat(1)和stat(2)。

如果你注意到socket(2)中“SEE ALSO”这部分，就会在其中发现一些我们将要讲到的系统调用。

(2) ri是Ruby命令行文档工具。Ruby安装程序会将安装核心库文档作为整个安装过程的一部分。

Ruby在某些方面缺乏良好的文档，不过我必须要说的是套接字库的文档相当全面。我们可以使用下面的命令来看看`Socket.new`的ri文档：

```
$ ri Socket.new
```

ri非常有用而且不需要连接互联网。如果你需要指南或者示例，它是不错的。

1.9 本章涉及的系统调用

每一章都会列出新介绍的系统调用，告诉你如何使用ri或手册页来获得其更多的信息。

- `Socket.new` → socket(2)

第2章

建立连接

TCP在两个端点之间建立连接。端点可能处于同一台主机，也可能位于不同的主机中。不管是哪一种情况，背后的原理都是一样的。

当你创建套接字时，这个套接字必须担任以下角色之一：(1) 发起者（initiator）；(2) 侦听者（listener）。两种角色必不可少。少了侦听套接字，就无法发起连接。没有连接的发起者，也就没必要进行侦听了。

在网络编程中，通常将从事侦听的套接字称作“服务器”，将发起连接的套接字称作“客户端”。下一章将观察它们各自的生命周期。

第3章

服务器生命周期

服务器套接字用于侦听连接而非发起连接，其典型的生命周期如下：

(1) 创建；

(2) 绑定；

(3) 侦听；

(4) 接受；

(5) 关闭。

我们已经讲过了“创建”。接下来继续讲解余下的部分。

3.1　服务器绑定

服务器生命周期中的第二步是绑定到监听连接的端口上。

```
                                    # ./code/snippets/bind.rb
require 'socket'

# 首先创建一个新的TCP套接字。
```

```
socket = Socket.new(:INET, :STREAM)

# 创建一个C结构体来保存用于侦听的地址。
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')

# 执行绑定。
socket.bind(addr)
```

这是一个低层次实现，演示了如何将TCP套接字绑定到本地端口上。实际上，它和用于实现同样功能的C代码几乎一模一样。

这个套接字现在被绑定到本地主机的端口4481上。其他套接字便不能再使用此端口，否则会产生异常Errno::EADDRINUSE。客户端套接字可以使用该端口号连接服务器套接字，并建立连接。

如果你运行上面的代码，会发现它立刻就会退出。这些代码倒是没错，但是还不足以侦听某个连接。接着往下阅读，看看如何将服务器置于侦听模式。

再次重述一遍，服务器需要绑定到某个特定的、双方商定的端口号上，客户端套接字随后会连接到该端口。

当然了，Ruby提供了语法上的便利，你无需直接使用Socket.pack_sockaddr_in或Socket#bind。不过在学习这些语法糖之前，我们很有必要避易就难地看看这一切是如何实现的。

3.1.1　该绑定到哪个端口

这对于每一个编写服务器的程序员而言都是一个非常重要的考量。应该选择随机端口吗？该如何知道是否已经有其他的程序将某个端口宣为己有？

任何在0~65 535之间的端口都可以使用，但是在选用之前别忘了一些重要的约定。

规则1：**不要使用0~1024之间的端口**。这些端口是作为熟知（well-known）端口并保留给系统使用的。例如HTTP默认使用端口80，SMTP默认使用端口25，rsync默认使用端口873。绑定到这些端口通常需要root权限。

规则2：**不要使用49 000~65 535之间的端口**。这些都是临时（ephemeral）端口。通常是由那些不需要运行在预定义端口，而只是需要一些端口作为临时之需的服务使用。它们也是后面所要讲到的连接协商（connection negotiation）过程的一部分。选择该范围内的端口可能会对一些用户造成麻烦。

除此之外，**1025~48 999之间端口的使用是一视同仁的**。如果你打算选用其中的一个作为服务器端口，那你应该看一下IANA的注册端口列表[①]，确保你的选择不会和其他流行的服务器冲突。

3.1.2 该绑定到哪个地址

在上面的例子中,我都是选择绑定到`0.0.0.0`,如果绑定到`127.0.0.1`或`1.2.3.4`，又会有什么不同呢？答案和你所使用的接口有关。

之前我提到过系统中有一个IP地址为`127.0.0.1`的环回接口。同时还会有另一个物理的、基于硬件的接口，使用不同的IP地址（假设是`192.168.0.5`）。当你绑定到某个由IP地址所描述的特定接口时，套接字就只会在该接口上进行侦听，而忽略其他接口。

① https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt.

如果绑定到127.0.0.1，那么你的套接字就只会侦听环回接口。在这种情况下，只有到localhost或127.0.0.1的连接才会被服务器套接字接受。环回接口仅限于本地连接使用，无法用于外部连接。

如果绑定到192.168.0.5，那么套接字只侦听此接口。任何寻址到这个接口的客户端都在侦听范围中，但是其他建立在localhost上的连接不会被该服务器套接字接受。

如果你希望侦听每一个接口，那么可以使用0.0.0.0。这样会绑定到**所有**可用的接口、环回接口等。大多数时候，这正是你所需要的。

```
                          # ./code/snippets/loopback_binding.rb
require 'socket'

# 该套接字将会绑定在环回接口，只侦听来自本地主机的客户端。
local_socket = Socket.new(:INET, :STREAM)
local_addr= Socket.pack_sockaddr_in(4481, '127.0.0.1')
local_socket.bind(local_addr)

# 该套接字将会绑定在所有已知的接口，侦听所有向其发送信息的客户端。
any_socket= Socket.new(:INET, :STREAM)
any_addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
any_socket.bind(any_addr)

# 该套接字试图绑定到一个未知的接口，结果导致Errno::EADDRNOTAVAIL。
error_socket = Socket.new(:INET, :STREAM)
error_addr= Socket.pack_sockaddr_in(4481, '1.2.3.4')
error_socket.bind(error_addr)
```

3.2　服务器侦听

创建套接字并绑定到特定端口之后，需要告诉套接字对接入的连接进

行侦听。

```
                                            # ./code/snippets/listen.rb
require 'socket'

# 创建套接字并绑定到端口4481。
socket = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
socket.bind(addr)

# 告诉套接字侦听接入的连接。
socket.listen(5)
```

和前一章代码唯一的不同之处就是在套接字上多了一个`listen`调用。

如果你运行这个代码片段，它仍旧会立刻退出。在服务器套接字能够处理连接之前，还需要另一个步骤。在下一章中我们会讲述。这里先解释下`listen`。

3.2.1 侦听队列

你可能注意到我们给`listen`方法传递了一个整数类型的参数。这个数字表示服务器套接字能够容纳的待处理（pending）的最大连接数。待处理的连接列表被称作侦听队列。

假设服务器正忙于处理某个客户端连接，如果这时其他新的客户端连接到达，将会被置于侦听队列。如果新的客户端连接到达且侦听队列已满，那么客户端将会产生`Errno::ECONNREFUSED`。

3.2.2 侦听队列的长度

侦听队列的长度听起来似乎是一个神奇的数字。为什么不把它设成10

000呢？为什么还要想着拒绝某个连接呢？这些问题提得很好。

我们首先来讨论一下侦听队列长度的限制。通过在运行时查看Socket::SOMAXCONN可以获知当前所允许的最大的侦听队列长度。在我的Mac上这个数字是128。我没法使用更大的数字。root用户可以在有需要的服务器上增加这个系统级别的限制。

假如你运行的服务器收到了错误信息Error::ECONNREFUSED，那增加侦听队列长度是一个不错的出发点。这也意味着你所服务的用户不得不等待服务器的响应。这也是提示你需要更多的服务器实例或是需要采用其他架构。

一般来说你肯定不希望拒绝连接，可以使用server.listen(Socket::SOMAXCONN)将侦听队列长度设置为允许的最大值。

3.3　接受连接

我们终于来到了服务器实际处理接入连接的环节。这是通过accept方法实现的。下面的代码演示了如何创建侦听套接字，接受首个连接：

```
                                # ./code/snippets/accept.rb
require 'socket'

# 创建服务器套接字。
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)

# 接受连接。
connection, _ = server.accept
```

如果现在运行这段代码，你会发现这次它没有**立刻**退出！没错，accept方法会一直阻塞到有连接到达。让我们用netcat发起一个连接：

```
$ echo ohai | nc localhost 4481
```

运行结果是nc(1)且Ruby程序都顺利退出。最精彩的并不在于此，而在于连接已经建立，一切工作正常。庆祝下吧！

3.3.1 以阻塞方式接受连接

accept调用是阻塞式的。在它接收到一个新的连接之前，它会一直阻塞当前线程。

> 还记得上一章讨论过的侦听队列吗？accept只不过就是将还未处理的连接从队列中弹出（pop）而已。如果队列为空，那么它就一直等，直到有连接被加入队列为止。

3.3.2 accept调用返回一个数组

在上面的例子中，我从accept调用中获得了两个返回值。accept方法实际上返回的是一个数组。这个数组包含两个元素：第一个元素是建立好的连接，第二个元素是一个Addrinfo对象。该对象描述了客户端连接的远程地址。

> **Addrinfo**
>
> Addrinfo是一个Ruby类，描述了一台主机及其端口号。它将端点信息进行了包装。你会在书中的其他地方看到它作为Socket接口的一部分出现。

> 可以使用Addrinfo.tcp('localhost', 4481)构建这些信息。一些有用的方法包括#ip_address和#ip_port。查看$ri Addrinfo了解更多信息。

接下来仔细查看一下#accept返回的连接和地址。

```
# ./code/snippets/accept_connection_class.rb
require 'socket'

# 创建服务器套接字。
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)

# 接受一个新连接。
connection, _ = server.accept

print 'Connection class: '
p connection.class

print 'Server fileno: '
p server.fileno

print 'Connection fileno: '
p connection.fileno

print 'Local address: '
p connection.local_address

print 'Remote address: '
p connection.remote_address
```

当服务器获得一个连接（使用之前用过的netcat命令），它会输出：

```
Connection class: Socket
Server fileno: 5
Connection fileno: 8
Local address: #<Addrinfo: 127.0.0.1:4481 TCP>
Remote address: #<Addrinfo: 127.0.0.1:58164 TCP>
```

代码输出告诉了我们一系列TCP连接相关的处理信息。下面逐一进行分析。

3.3.3 连接类

尽管`accept`返回了一个"连接"，但是这段代码告诉我们并没有特殊的连接类（connection class）。一个连接实际上就是`Socket`的一个实例。

3.3.4 文件描述符

我们知道`accept`返回一个`Socket`的实例，不过这个连接的文件描述符编号和服务器套接字不一样。文件描述符编号是内核用于跟踪当前进程所打开文件的一种方法。

套接字是文件吗？

套接字是文件。至少在Unix世界中，所有的一切都被视为文件。[①] 这包括文件系统中的文件以及管道、套接字和打印机，等等。

这表明`accept`返回了一个不同于服务器套接字的全新`Socket`。这个`Socket`实例描述了特定的连接。这一点很重要。每个连接都由一个全新的`Socket`对象描述，这样服务器套接字就可以保持不变，不停地接受新的连接。

① http://ph7spot.com/musings/in-unix-everything-is-a-file.

3.3.5　连接地址

连接对象知道两个地址：本地地址和远程地址。其中的远程地址是accept的第二个返回值，不过也可以从连接中的remote_address访问到。

连接的local_address指的是本地主机的端点，remote_address指的是另一端的端点，而这个端点可能位于另一台主机，也可能存在于同一台主机（本例便是如此）。

每一个TCP连接都是由“本地主机、本地端口、远程主机、远程端口”这组唯一的组合所定义的。对于所有TCP连接而言，这4个属性的组合必须是唯一的。

让我们来考虑一下。你可以同时从本地主机上向远程主机发起两个连接，只要远程端口不重复即可。类似地，如果远程端口不重复，你也可以在同一个本地端口上同时接受来自远程主机的两个连接。但是如果两组本地端口和远程端口都是一样的，那就无法同时建立到同一台远程主机的两个连接了。

3.3.6　accept循环

accept会返回一个连接。在前面的代码中，服务器接受了一个连接后退出。在编写真正的服务器代码时，只要还有接入的连接，我们肯定希望不停地侦听。这可以很轻松地通过循环来实现：

```
                         # ./code/snippets/naive_accept_loop.rb
require 'socket'

# 创建服务器套接字。
server = Socket.new(:INET, :STREAM)
```

```
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)

# 进入无限循环，接受并处理连接。
loop do
  connection, _ = server.accept
  # 处理连接。
  connection.close
end
```

这是使用Ruby编写某类服务器的一种常见方法。由于这种方式在实际中应用广泛，Ruby在其基础上提供了一些语法糖。在本章结尾处我们会看到一些经由Ruby包装过的方法。

3.4 关闭服务器

一旦服务器接受了某个连接并处理完毕，那么最后一件事就是关闭该连接。这就算是完成了一个连接的“创建–处理–关闭”的生命周期。

我就不再贴上另一段代码了，继续参考上面的代码段。在接受新的连接之前，先在之前的连接上调用close即可。

3.4.1 退出时关闭

为什么需要close？当程序退出时，系统会帮你关闭所有打开的文件描述符（包括套接字）。那为什么还要自己动手去关闭呢？这有以下两个很好的理由。

(1) 资源使用。如果你使用了套接字却没有关闭它，那么那些你已不再使用的套接字的引用很可能依然保留着。在Ruby中，垃圾收集

器可是你的好伙伴，帮你清理用不着的连接，不过保持自己所用资源的完全控制权，丢掉不再需要的东西总是一个不错的想法。要注意的是，垃圾收集器会将它收集到的所有一切全部关闭。

(2) 打开文件的数量限制。这实际上是上一个理由的延伸。所有进程都只能够打开一定数量的文件。还记不记得前面说过每个连接都是一个文件？保留无用的连接会使进程逐步逼近这个上限，迟早会出问题。

> 要获知当前进程所允许打开文件的数量，你可以使用Process.getrlimit(:NOFILE)。返回值是一个数组，包含了软限制（用户配置的设置）和硬限制（系统限制）。
>
> 如果想将限制设置到最大值，可以使用Process.setrlimit(Process.getrlimit(:NOFILE)[1])。

3.4.2 不同的关闭方式

考虑到套接字允许双向通信（读/写），实际上可以只关闭其中一个通道。

```ruby
                               # ./code/snippets/close_write.rb
require 'socket'

# 创建服务器套接字。
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)
connection, _ = server.accept
```

```
# 该连接随后也许不再需要写入数据，但是可能仍需要进行读取。
connection.close_write

# 该连接不再需要进行任何数据读写操作。
connection.close_read
```

关闭写操作流（write stream）会发送一个EOF到套接字的另一端。（我们很快就会讲到EOF。）

close_write和close_read方法在底层都利用了shutdown(2)。同close(2)明显不同的是：即便是存在着连接的副本，shutdown(2)也可以完全关闭该连接的某一部分。

连接副本是怎么回事？

可以使用Socket#dup创建文件描述符的副本。这实际上是在操作系统层面上利用dup(2)复制了底层的文件描述符。不过这种情况极为罕见，你不大可能会碰上。

获得一个文件描述符副本的更常见的方法是利用Process.fork方法。该方法创建了一个全新的进程（仅在Unix环境中），这个进程和当前进程一模一样。除了拥有当前进程在内存中的所有内容之外，新进程还通过dup(2)获得了所有已打开的文件描述符的副本。

close会关闭调用它的套接字实例。如果该套接字在系统中还有其他副本，那么这些副本不会被关闭，所占用的资源也不会被回收。没错，连接的其他副本仍然可以交换数据，即便是在某个实例已经被关闭的情况下。

和close不同，shutdown会完全关闭在当前套接字及其副本上的通信。但是它并不会回收套接字所使用过的资源。每个套接字实例仍必

须使用close结束它的生命周期。

```
                                   # ./code/snippets/shutdown.rb
require 'socket'

# 创建服务器套接字。
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)
connection, _ = server.accept

# 创建连接副本。
copy = connection.dup

# 关闭所有连接副本上的通信。
connection.shutdown

# 关闭原始连接。副本会在垃圾收集器进行收集时关闭。
connection.close
```

3.5 Ruby 包装器

我们都熟知并热爱Ruby所提供的优雅语法，它用来创建及使用服务器套接字的扩展也会让你爱不释手。这些便捷的方法将样本代码（boilerplate code）包装在定制的类中并尽可能地利用Ruby的语句块。下面我们来看一下它是如何实现的。

3.5.1 服务器创建

首先是TCPServer类。它将进程中“服务器创建”这部分进行了非常简洁的抽象。

```
                        # ./code/snippets/server_easy_way.rb
require 'socket'

server = TCPServer.new(4481)
```

瞧，现在看起来更有Ruby味儿了。这段代码实际上是如下代码的替换：

```
                        # ./code/snippets/server_hard_way.rb
require 'socket'

server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(5)
```

我很清楚自己该选哪一种！

> 创建一个TCPServer实例返回的实际上并不是Socket实例，而是TCPServer实例。两者的接口几乎一样，但还是存在一些重要的差异。其中最明显的就是TCPServer#accept只返回连接，而不返回remote_address。
>
> 有没有注意到我们并没有为这些构造函数指定侦听队列的长度？因为用不着使用Socket::SOMAXCONN，Ruby默认将侦听队列长度设置为5。如果需要更长的侦听队列，可以调用TCPServer#listen。

随着IPv6的发展势头越发变得迅猛，你的服务器也许得能够同时处理IPv4和IPv6。使用这个Ruby包装器会返回两个TCP套接字，一个可以通过IPv4连接，另一个可以通过IPv6连接，两者都在同一个端口上进行侦听。

```
                              # ./code/snippets/server_sockets.rb
require 'socket'

servers = Socket.tcp_server_sockets(4481)
```

3.5.2　连接处理

除了创建服务器，Ruby也为连接处理提供了优美的抽象。

还记得使用loop处理多个连接吗？聪明人才不会用loop呢。应该像下面这样做：

```
                              # ./code/snippets/accept_loop.rb
require 'socket'

#创建侦听套接字。
server = TCPServer.new(4481)

# 进入无限循环接受并处理连接。
Socket.accept_loop(server) do |connection|
  # 处理连接。
  connection.close
end
```

要注意连接并不会在每个代码块结尾处自动关闭。传递给代码块的参数和accept调用的返回值一模一样。

Socket.accept_loop还有另外一个好处：你可以向它传递多个侦听套接字，它可以接受在这些套接字上的全部连接。这和Socket.tcp_server_sockets可谓是相得益彰：

```
                    # ./code/snippets/accept_server_sockets.rb
require 'socket'
```

```
# 创建侦听套接字。
servers = Socket.tcp_server_sockets(4481)

# 进入无限循环，接受并处理连接。
Socket.accept_loop(servers) do |connection|
  # 处理连接。
  connection.close
end
```

我们传递了多个套接字给Socket.accept_loop，由它来进行妥善处理。

3.5.3　合而为一

这些Ruby包装器的集大成者是Socket.tcp_server_loop，它将之前的所有步骤合而为一：

```
                                 # ./code/snippets/tcp_server_loop.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # 处理连接。
  connection.close
end
```

该方法实际上只是Socket.tcp_server_sockets和Socket.accept_loop的一个包装器而已，但再也没有比它更简洁的写法了。

3.6　本章涉及的系统调用

- Socket#bind → bind(2)
- Socket#listen → listen(2)
- Socket#accept → accept(2)

- `Socket#local_address` → getsockname(2)
- `Socket#remote_address` → getpeername(2)
- `Socket#close` → close(2)
- `Socket#close_write` → shutdown(2)
- `Socket#shutdown` → shutdown(2)

第4章

客户端生命周期

我之前提到过一个网络连接有两个重要的组成部分。服务器负责侦听及处理接入的连接。客户端负责向服务器发起连接，也就是说，它知道特定服务器的位置并创建指向外部服务器的连接。

很显然，没有客户端的服务器是不完整的。

客户端的生命周期要比服务器短一些。它包括以下几个阶段：

(1) 创建；

(2) 绑定；

(3) 连接；

(4) 关闭。

第一个阶段对于客户端和服务器来说都是一样的，所以就客户端而言，我们从第二个阶段“绑定”开始讲起。

4.1　客户端绑定

客户端套接字和服务器套接字一样，都是以 bind 作为起始。在服务器部分，我们使用特定的地址和端口来调用 bind。很少会有服务器不去调用 bind，**也很少会有客户端去调用** bind。如果客户端套接字（或者服务器套接字）不调用 bind，那么它会从临时端口范围内获得一个随机端口号。

> **为什么不调用bind?**
>
> 客户端之所以不需要调用bind，是因为它们无需通过某个已知端口访问。而服务器要绑定到特定端口的原因是，客户端需要通过特定的端口访问到服务器。
>
> 以FTP为例。它的熟知端口是21。因此FTP服务器应该绑定到该端口，这样客户端就知道从哪里获取FTP服务了。客户端可以从任何端口发起连接，客户端选择的端口号不会影响到服务器。
>
> 客户端不需要调用bind，因为没有人需要知道它们的端口号。

这一节并没有展示什么代码，因为我的建议是：不要给客户端绑定端口！

4.2　客户端连接

客户端和服务器真正的区别就在于 connect 调用。该调用发起到远程套接字的连接。

```
                                  # ./code/snippets/connect.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)

# 发起到google.com 端口 80 的连接。
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
socket.connect(remote_addr)
```

因为我们在这里使用的是低层次函数，所以需要将地址对象转换成 C 语言结构体的描述形式。

该代码片段从本地的临时端口向在 google.com 的端口 80 上进行侦听的套接字发起 TCP 连接。注意我们并没有调用 bind。

连接故障

在客户端的生命周期中，很可能在服务器还没准备好接受连接之前客户端就发起了连接。同样也有可能连接了一个并不存在的服务器。两种情况实际上是殊途同归。因为 TCP 所具备的容错性，它会尽最大可能等待远程主机的回应。

下面试试连接一个不可用的端点：

```
                     # ./code/snippets/connect_non_existent.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)

# 尝试在gopher port①上连接google.com。
remote_addr = Socket.pack_sockaddr_in(70, 'google.com')
```

① http://en.wikipedia.org/wiki/Gopher_(protocol).——译者注

```
socket.connect(remote_addr)
```

如果你运行这段代码，它花费很长时间才能从 connect 调用返回。conncet 调用默认有一段较长时间的超时。

这对于那些带宽有限、需要花费长时间建立连接的客户端来说是有意义的。至于那些没有带宽烦恼的客户端，这种默认的超时行为也无碍于同远端快速建立连接。

但如果出现超时，最终会产生一个 Errno::ETIMEOUT 异常。这属于一般性的超时异常，如果同套接字打交道，就表明所请求的操作超时了。如果你对调校套接字超时感兴趣，请参看第 15 章。

当客户端连接到一个已经调用过 bind 和 listen，但尚未调用 accept 的服务器时，也会出现同样的情况。只有远程服务器接受了连接，connect 调用才会成功返回。

4.3　Ruby 包装器

创建客户端套接字的代码几乎和创建服务器套接字一样繁琐、低级。如我们所愿，Ruby 也对其进行了包装，使它们更易于使用。

客户端创建

在向你展现优美的 Ruby 风格的代码之前，我要先让你看看那些低级繁琐的代码，这样才能够有所比较：

```
                                # ./code/snippets/connect.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)

# 发起到 google.com 端口 80 的连接。
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
socket.connect(remote_addr)
```

用了语法糖之后：

```
                        # ./code/snippets/client_easy_way.rb
require 'socket'

socket = TCPSocket.new('google.com', 80)
```

这下子感觉好多了。之前的三行代码、两个构造函数以及大量的上下文被精简为一个构造函数。

还有一个使用 Socket.tcp 的类似的客户端构建方法，它可以采用代码块的形式：

```
                      # ./code/snippets/client_block_form.rb
require 'socket'

Socket.tcp('google.com', 80) do |connection|
  connection.write"GET / HTTP/1.1\r\n"
  connection.close
end

# 如果省略代码块参数，则行为方式同 TCPSocket.new()一样。
client = Socket.tcp('google.com', 80)
```

4.4　本章涉及的系统调用

- `Socket#bind` → bind(2)
- `Socket#connect` → connect(2)

第5章

交换数据

前面的部分都是关于建立连接，连接两个端点的内容。尽管这本身也挺有意思，但是如果连接上没有数据的交换，你实际上什么有意义的事也做不了。本章就来解决这个问题。最终我们不仅可以建立服务器和客户端的连接，还能够让它们进行数据交换。

在深入学习之前，我想强调，你可以将 TCP 连接想象成一串连接了本地套接字和远程套接字的管子，我们可以沿着这根管子发送、接收数据。这种想象有助于增进我们对 TCP 的理解。Berkeley 套接字 API 就是这样子设计的，我们同样以这种方式对身边的世界进行建模，解决各类问题。

在实际中，所有的数据都被编码为 TCP/IP 分组，在抵达终点的路上可能会途经多台路由器和主机。这个世界有点疯狂，所以最好记住并非事事都会一帆风顺，不过我们也要感谢在这个疯狂的世界中辛勤工作的人们，他们为我们掩盖了那些不如意之处，使我们保留了对于世界的简单想象。

流

还有一件事我需要说清楚，即 TCP 所具有的基于流的性质，这一点我们还没有讲过。

回到本书伊始，当时我们创建了第一个套接字并传入了一个叫做`:STREAM`的选项，该选项表明我们希望使用一个流套接字。TCP 是一个基于流的协议。如果我们在创建套接字时没有传入`:STREAM`选项，那就无法创建 TCP 套接字。

那么这究竟意味着什么？这对于我们的代码会产生什么影响呢？

首先，前面提到的术语“分组”其实已经给出了暗示。从协议层面上而言，TCP 在网络上发送的是分组。

不过我们不打算讨论分组。从应用程序代码的角度上来说，TCP 连接提供了一个不间断的、有序的通信流。只有流，别无其他。

让我们用一些伪代码进行演示。

```
# 下面的代码会在网络上发送 3 份数据，一次一份。
data = ['a', 'b', 'c']

for piece in data
  write_to_connection(piece)
end

# 下面的代码在一次操作中读取全部数据。
result = read_from_connection #=> ['a', 'b', 'c']
```

这段代码想传达的是：*流并没有消息边界的概念*。即便是客户端分别发送了 3 份数据，服务器在读取时，也是将其作为一份数据来接收。它并不知道客户端是分批发送的数据。

要注意的是，尽管并不保留消息边界，但是流中内容的次序还是会保留的。

第6章

套接字读操作

至此我们已经讨论了不少关于连接的话题。现在我们要进入真正有趣的部分：如何在套接字连接上传送数据。在使用套接字时，有多种方法可以进行数据读写，这没什么好惊讶的。除此之外，Ruby还为我们提供了一些优雅便捷的包装器。

本章将深入学习各种读取数据的方法以及各自的适用场景。

6.1　简单的读操作

从套接字读取数据最简单的方法是使用read：

```
                                    # ./code/snippets/read.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # 从连接中读取数据最简单的方法。
  puts connection.read

  # 完成读取之后关闭连接。让客户端知道不用再等待数据返回。
  connection.close
end
```

如果你在终端上运行这个例子，在另一个终端上运行下面的netcat命令，将会在Ruby服务器中看到输出结果：

```
$ echo gekko | nc localhost 4481
```

> 如果你使用过Ruby的File API，那么这段代码可能看起来挺眼熟。Ruby的各种套接字类以及File在IO中都有一个共同的父类。Ruby中所有的IO对象（套接字、管道、文件……）都有一套通用的接口，支持read、write、flush等方法。
>
> 这的确不是Ruby的创新。底层的read(2)，write(2)等系统调用都可以作用于文件、套接字、管道等之上。这种抽象源自于操作系统核心本身。记住，**一切皆为文件**。

6.2 没那么简单

读取数据的方法很简单，但是容易出错。如果你运行下面的netcat命令，然后撒手不管，服务器将永远不会停止读取数据，也永远不会退出：

```
$ tail -f /var/log/system.log | nc localhost 4481
```

造成这种情况的原因是EOF（end-of-file）。下一节我们会详细讨论。此刻我们暂且不理会EOF，来看一种不太成熟的解决方法。

这个问题的关键在于tail -f根本就不会停止发送数据。如果tail没有数据可以发送，它会一直等到有为止。这使得连接netcat的管道一直处于打开状态，因此netcat也永远都不会停止向服务器发送数据。

服务器的read调用就一直被阻塞着，直到客户端发送完数据为止。在我们的这个例子中，服务器就这样等待……等待……一直等待……，在这期间它会将接收到的数据缓冲起来，不返回给应用程序。

6.3　读取长度

解决以上问题的一个方法是指定最小的读取长度。这样就不用等到客户端结束发送才停止读取操作，而是告诉服务器读取（read）特定的数据量，然后返回。

```
                              # ./code/snippets/read_with_length.rb
require 'socket'
one_kb = 1024 # 字节数

Socket.tcp_server_loop(4481) do |connection|
  # 以为1KB为单位进行读取。
  while data = connection.read(one_kb) do
    puts data
  end

  connection.close
end
```

和上一节一样运行以下命令：

```
$ tail -f /var/log/system.log | nc localhost 4481
```

上面的代码会使服务器在netcat命令运行的同时，以1KB为单位打印出数据。

这个例子的不同之处在于我们给read传递了一个整数。它告诉read

在读取了一定数量的数据后就停止读取并返回。由于希望得到所有可用的数据，我们在调用read方法时使用了循环，直到它不再返回数据为止。

6.4 阻塞的本质

read调用会一直阻塞，直到获取了完整长度（full length）的数据为止。在上面的例子中每次读取1KB。运行几次之后，应该会清楚地发现：如果读取了一部分数据，但是不足1KB，那么read会一直阻塞，直至获得完整的1KB数据为止。

采用这种方法实际上有可能会导致死锁。如果服务器试图从连接中读取1KB的数据，而客户端只发送了500B后就不再发送了，那么服务器就会一直傻等着那没发的500B！

有两种方法可以补救：(1) 客户端发送完500B后再发送一个EOF；(2) 服务器采用部分读取（partial read）的方式。

6.5 EOF 事件

当在连接上调用read并接收到EOF事件时，就可以确定不会再有数据，可以停止读取了。这个概念对于理解IO操作至关重要。

先插点历史典故：EOF代表“end of file”（文件结束）。你大概会说：“我们现在处理的又不是文件……。”说的基本没错，不过请记住“一切皆是文件”。

有时候你可能会看到“EOF字符”的说法，不过其实并没有这样的东西。EOF并不代表某种字符序列，它更像是一个状态事件（state event）。如果一个套接字没有数据可写，它可以使用shutdown或close来表明自己不再需要写入任何数据。这就会导致一个EOF事件被发送给在另一端进行读操作的进程，这样它就知道不会再有数据到达了。

让我们从头再来审视并解决上一节的那个问题：如果服务器希望接受1KB数据，而客户端却只发送了500B。

一种改进方法是客户端发送500B，然后再发送一个EOF事件。服务器接收到该事件后就停止读取，即便是还未接收够1KB。EOF表明不会再有数据到达了。

下面是正确的数据读取代码：

```
                            # ./code/snippets/read_with_length.rb
require 'socket'
one_kb = 1024 # 字节数

Socket.tcp_server_loop(4481) do |connection|
  # 一次读取1KB的数据。
  while data = connection.read(one_kb) do
    puts data
  end

  connection.close
end
```

客户端连接：

```
                            # ./code/snippets/write_with_eof.rb
require 'socket'

client = TCPSocket.new('localhost', 4481)
```

```
client.write('gekko')
client.close
```

客户端发送EOF最简单的方式就是关闭自己的套接字。如果套接字已经关闭，肯定不会再发送数据了！

> 要说起EOF，它的名字还是挺恰如其分的。当你调用File#read时（同Socket#read的行为方式类似），它会一直进行数据读取，直到没有数据为止。一旦读完整个文件，它会接收到一个EOF事件并返回已读取到的数据。

6.6 部分读取

之前我提到过一个术语“部分读取”。这也是一种解决方案，接下来我们就来介绍一下。

我们刚刚介绍过的那种读取数据的方法算是种懒办法。当你调用read时，在返回数据之前它会一直等待，直到获得所需要的最小长度或是EOF。还有另外一种反其道而行的读取方法。那就是readpartial。

readpartial并不会阻塞，而是立刻返回可用的数据。调用readpartial时，你必须传递一个整数作为参数，来指定最大的长度。readpartial最多读取到指定长度。如果你指明读取1KB数据，但是客户端只发送了500B，readpartial并不会阻塞。它会立刻将已读取到的数据返回。

在服务器端运行：

```
                    # ./code/snippets/readpartial_with_length.rb
require 'socket'
one_hundred_kb = 1024 * 100

Socket.tcp_server_loop(4481) do |connection|
  begin
    # 每次读取100KB或更少。
    while data = connection.readpartial(one_hundred_kb) do
      puts data
    end
  rescue EOFError
  end

  connection.close
end
```

结合以下客户端命令：

```
$ tail -f /var/log/system.log | nc localhost 4481
```

从中可以看到服务器会持续读取一切可用的数据，而不是非要等足100KB。只要有数据，readpartial就会将其返回，即便是小于最大长度。

就EOF而言，readpartial的工作方式不同于read。当接收到EOF时，read仅仅是返回，而readpartial则会产生一个EOFError异常，提醒我们要留心。

再总结一下：read很懒惰，只会傻等着，以求返回尽可能多的数据。相反，readpartial更勤快，只要有可用的数据就立刻将其返回。

在学习过write之后，我们会转向缓冲区。到那个时候，我们就可以回答一些有意思的问题了，例如：我应该一次性读取多少数据？小读

取量的多次读操作和大读取量的单次读操作究竟哪一种更好？

6.7　本章涉及的系统调用

- `Socket#read` → read(2)，行为类似fread(3)
- `Socket#readpartial` → read(2)

第7章

套接字写操作

我知道一些读者已经想到了：一个套接字要想读取数据，另一个套接字就必须写入数据！恭喜你，答对了！

只有一种方法可以向套接字中写入数据，那就是`write`方法。它的用法非常直观。跟着直觉做就行了。

```
                                       # ./code/snippets/write.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # 向连接中写入数据的最简单的方法。
  connection.write('Welcome!')
  connection.close
end
```

除了`write`调用之外就没什么好说的了。在下一章学习缓冲区的时候，我们将会解答一些有趣的问题。

本章涉及的系统调用

- `Socket#write` → write(2)

第8章

缓　　冲

在本章中，我们会解答几个重要的问题：在一次调用中应该读/写多少数据？如果`write`成功返回，是否意味着连接的另一端已经接收到了数据？是否应该将一个大数据量的`write`分割成多个小数据量进行多次写入？这样会造成怎样的影响？

8.1　写缓冲

我们先来讨论在TCP连接上调用`write`进行写入的时候究竟发生了什么？

当你调用`write`并返回时，就算是没有引发异常，也并不代表数据已经通过网络顺利发送并被客户端套接字接收到。`write`返回时，它只是表明你已经将数据提交给了Ruby的IO系统和底层的操作系统内核。

在应用程序代码和实际的网络硬件之间至少还存在一个缓冲层。让我们先来指明它们的具体所在，然后再来看看如何同它们打交道。

如果`write`成功返回，这仅能保证你的数据已经交到了操作系统内核的手中。它可以立刻发送数据，也可以出于效率上的考虑暂不发送，将其同别的数据进行合并。

TCP套接字默认将`sync`设置为`true`。这就跳过了Ruby的内部缓冲[①]，否则就又要多出一个缓冲层了。

> **为何需要缓冲区？**
>
> 所有的IO缓冲都是出于性能的考虑，它们通常能够显著提高性能。
>
> 通过网络发送数据的速度很慢[②]，相当慢。缓冲使得`write`调用可以立刻返回。然后在幕后由内核将所有还未执行的写操作汇总到一起，在发送时进行分组及优化，在实现最佳性能的同时避免网络过载。在网络层面上，发送大量的小分组会引发可观的开销，因此内核会将多个小数据量的写操作合并成较大数据量的写操作。

8.2 该写入多少数据

鉴于目前对于缓冲的了解，我们再次摆出这个问题：是应该采用多个小数据量的`write`调用还是单个大数据量的`write`调用？

幸好有缓冲区，我们其实无需考虑这个问题。通常你只需要将用到的数据一股脑地写入就行了，由内核通过对写操作进行分割或合并来调节性能，如此一来便能获得不错的效果。如果你要做一个相当大数据

① http://jstorimer.com/2012/09/25/ruby-io-buffers.html.

② https://gist.github.com/2841832.

量的`write`，比如文件或大数据写入，那最好是将这些数据进行分割，避免全部载入内存中。

通常情况下，获得最佳性能的方法是**一口气写入所有的数据**，让内核决定如何对数据进行结合。显然，唯一需要做的是优化你的应用程序。

8.3　读缓冲

不止是写操作，读操作同样会被缓冲。

如果你调用`read`从TCP连接中读取数据并给它传递一个最大的读取长度，Ruby实际上可能会接收大于你指定长度的数据。

在这种情况下，“多出的”数据会被存储在Ruby内部的读缓冲区中。在下次调用`read`时，Ruby会先查看自己的内部缓冲区中有没有未读取的数据，然后再通过操作系统内核请求更多的数据。

8.4　该读取多少数据

这个问题的答案可不像写缓冲那样直观，我们来看看涉及的问题以及最佳实践。

因为TCP提供的是数据流，我们无法得知发送方到底发送了多少数据。这就意味着在决定读取长度的时候，我们只能靠猜测。

为什么不指定一个很大的读取长度来确保总是可以得到所有可用的数据呢？当指定读取长度时，内核会为我们分配一定的内存。如果用不着那么多，就会造成资源浪费。

如果我们指定一个较小的读取长度，它需要多次才能够读取完全部的数据。这会导致每次系统调用所引发的大量开销问题。

所以如同大多数事情一样，如果你根据应用程序所要接收的数据大小来进行调优，那么就能获得最佳的性能。如果要接收大量的大数据块怎么办？那你可能得指定一个较大的读取长度。

这个问题并没有万能解药，不过我偷了点懒，直接去研究了一下使用了套接字的各类Ruby项目，看看它们是如何在该问题上达成共识的。

我看过Mongrel、Unicorn、Puma、Passenger以及Net::HTTP，它们无一例外地采用了`readpartial(1024*16)`。所有这些Web项目都是用16KB作为各自的读取长度。

然而，redis-rb使用1KB作为读取长度。

你总是可以通过调优服务器来适应当下的数据量以获得最佳的性能，在犹豫不定时，16KB是一个公认比较合适的读取长度。

第9章

第一个客户端/服务器

前面几章讲述了如何建立连接以及大量有关数据交换的内容。到目前为止我们基本上都是和一些小型的、自包含的代码片段小打小闹，接下来该运用学到的知识编写一个网络服务器和客户端了。

9.1 服务器

就这个服务器而言，我们打算编写一种全新的NoSQL解决方案。它将作为Ruby散列表之上的一个网络层。我们称其为CloudHash。

下面是这个简单的CloudHash服务器的完整实现：

```
                                    # ./code/cloud_hash/server.rb
require 'socket'

module CloudHash
  class Server
    def initialize(port)
      # 创建底层的服务器套接字。
      @server = TCPServer.new(port)
      puts "Listening on port #{@server.local_address.ip_port}"
      @storage = {}
```

```
    end

    def start
      # accept循环。
      Socket.accept_loop(@server) do |connection|
        handle(connection)
        connection.close
      end
    end

    def handle(connection)
      # 从连接中进行读取，直到出现EOF。
      request = connection.read

      # 将hash操作的结果写回。
      connection.write process(request)
    end

    # 所支持的命令：
    # SET key value
    # GET key
    def process(request)
      command, key, value = request.split
      case command.upcase
      when 'GET'
        @storage[key]

      when 'SET'
        @storage[key] = value
      end
    end
  end
end

server = CloudHash::Server.new(4481)
server.start
```

9.2 客户端

下面是客户端的完整实现：

```
                                  # ./code/cloud_hash/client.rb
require 'socket'

module CloudHash
  class Client
    class << self
      attr_accessor :host, :port
    end

    def self.get(key)
      request "GET #{key}"
    end

    def self.set(key, value)
      request "SET #{key} #{value}"
    end

    def self.request(string)
      # 为每一个请求操作创建一个新连接。
      @client = TCPSocket.new(host, port)
      @client.write(string)

      # 完成请求之后发送EOF。
      @client.close_write

      # 一直读取到EOF来获取响应信息。
      @client.read
    end
  end
end

CloudHash::Client.host = 'localhost'
CloudHash::Client.port = 4481
```

```
puts CloudHash::Client.set 'prez', 'obama'
puts CloudHash::Client.get 'prez'
puts CloudHash::Client.get 'vp'
```

9.3 投入运行

接下来把它们组装起来投入运行吧！

启动服务器：

```
$ ruby code/cloud_hash/server.rb
```

别忘了其中的数据结构就是一个散列表。运行客户端将会执行以下操作：

```
$ tail -4 code/cloud_hash/client.rb
puts CloudHash::Client.set 'prez', 'obama'
puts CloudHash::Client.get 'prez'
puts CloudHash::Client.get 'vp'

$ ruby code/cloud_hash/client.rb
```

9.3 分析

我们前面都做了些什么？我们使用网络API将Ruby散列表进行了包装，不过并没有包装全部的Hash API，而仅仅是读写器（getter/setter）而已。代码中的不少地方都是些网络编程的样板代码，所以应该很容易就能看出该如何扩展这个例子，以便涵盖更多的Hash API。

我对代码作了注释，便于你搞明白程序的来龙去脉，此外我还坚持贯

彻了我们已经学习过的那些概念，比如建立连接、EOF等。

但总的来说，CloudHash还比较粗糙。前几章我们学习过连接建立和数据交换的基础知识。在这里这些内容都得到了应用。没有涉及的内容是有关架构模式的最佳实践、设计方法以及一些还没学到的高级特性。

比如说，有没有注意到客户端必须为每一个发送的请求发起一个新的连接？如果你想连续发送一批请求，那么每个请求都会占用一个连接。我们目前的服务器设计要求必须如此。它处理一条来自客户端套接字的命令，然后关闭。

并非一定要采用这种处理方式。建立连接会引发开销，CloudHash完全可以在同一个连接上处理多个请求。

有几种改进的方法。客户端/服务器可以使用一种不需要发送EOF来分隔消息的简单消息协议进行通信。这样可以在单个连接上发送多个请求，而服务器仍旧是依次处理每个客户端连接。如果某个客户端要发送大量请求或者长时间保持连接，那么其他客户端就无法同服务器进行交互了。

我们可以在服务器中加入某种形式的并发来解决这个问题。本书余下的部分将以目前你学到的内容作为基础，致力于帮助你编写出高效、易于理解、功能完善的网络程序。就CloudHash本身而言，并没有很好地展示如何进行套接字编程。

第10章

套接字选项

我们以套接字选项作为这一系列有关套接字高级技术的章节的开始。套接字选项是一种配置特定系统下套接字行为的低层手法。因为涉及低层设置，所以Ruby并没有为这方面的系统调用提供便捷的包装器。

10.1 SO_TYPE

我们先来看看如何获得套接字类型这一套接字选项。

```
                              # ./code/snippets/getsockopt.rb
require 'socket'

socket = TCPSocket.new('google.com', 80)
# 获得一个描述套接字类型的Socket::Option实例。
opt = socket.getsockopt(Socket::SOL_SOCKET, Socket::SO_TYPE)

# 将描述该选项的整数值同存储在Socket::SOCK_STREAM中的整数值进行比较。
opt.int == Socket::SOCK_STREAM  #=> true
opt.int == Socket::SOCK_DGRAM  #=> false
```

getsockopt返回一个Socket::Option实例。在这个层面上进行操作时，所有一切都被转化成了整数。因此SocketOption#int可以获得

与返回值相关联的底层的整数值。

在这里我获得的是套接字类型（记得吧，创建第一个套接字时就指定了这个类型），所以将int值同各种Socket类型常量进行比较，结果发现这是一个STREAM套接字。

记住Ruby总是会提供便于记忆的符号来代替这些常量。上面的代码也可以写成这样：

```
                        # ./code/snippets/getsockopt_wrapper.rb
require 'socket'

socket = TCPSocket.new('google.com', 80)
# 使用符号名而不是常量。
opt = socket.getsockopt(:SOCKET, :TYPE)
```


10.2 SO_REUSE_ADDR

这是每个服务器都应该设置的一个常见选项。

SO_REUSE_ADDR选项告诉内核：如果服务器当前处于TCP的TIME_WAIT状态，即便另一个套接字要绑定（bind）到服务器目前所使用的本地地址也无妨。

TIME_WAIT状态

当你关闭（close）了某个缓冲区，但其中仍有未处理数据的套接字之时就会出现TIME_WAIT状态。前面曾说过，调用write只是保证数据已经进入了缓冲层。当你关闭一个套接字时，它未处理的数据并不会被丢弃。

在幕后，内核使连接保持足够长的打开时间，以便将未处理的数据发送完毕。这就意味着它必须发送数据，然后等待接收方的确认，以免数据需要重传。

如果关闭一个尚有数据未处理的服务器并立刻将同一个地址绑定到另一个套接字上（比如重启服务器），则会引发一个Errno::EADDRINUSE，除非未处理的数据被丢弃掉。设置SO_REUSE_ADDR可以绕过这个问题，使你可以绑定到一个处于TIME_WAIT状态的套接字所使用的地址上。

下面是打开该选项的方法：

```
                                # ./code/snippets/reuseaddr.rb
require 'socket'

server = TCPServer.new('localhost', 4481)
server.setsockopt(:SOCKET, :REUSEADDR, true)

server.getsockopt(:SOCKET, :REUSEADDR)  #=> true
```

注意，TCPServer.new、Socket.tcp_server_loop及其类似的方法默认都打开了此选项。

可以通过setsockopt(2)查看系统上可用的套接字选项的完整列表。

10.3　本章涉及的系统调用

- Socket#setsockopt → setsockopt(2)
- Socket#getsockopt → getsockopt(2)

第11章

非阻塞式IO

本章是关于非阻塞式IO的内容。它与异步式或事件驱动式IO不同。如果你还不了解其中的差别，那么随着书中后面部分的学习，一切都会变得清晰。

非阻塞式IO同下一章要介绍的连接复用之间的关系非常紧密，不过我打算先讲述前者，因为非阻塞式IO本身就已经能独当一面了。

11.1　非阻塞式读操作

还记得我们之前学过的`read`吗？我提到过`read`会一直保持阻塞，直到接收到EOF或是获得指定的最小字节数为止。如果客户端没有发送EOF，就可能会导致阻塞。这种状况可以通过`readpartial`暂时解决，`readpartial`会立即返回所有的可用数据。但如果没有数据可用，那么`readpartial`仍旧会陷入拥塞。如果需要一种不会阻塞的读操作，可以使用`read_nonblock`。

和`readpartial`非常类似，`read_nonblock`需要一个整数的参数，指定需要读取的最大字节数。记住`read_nonblock`和`readpartial`一

样，如果可用的数据小于最大字节数，那就只返回可用数据。代码演示如下：

```
                                  # ./code/snippets/read_nonblock.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  loop do
    begin
      puts connection.read_nonblock(4096)
    rescue Errno::EAGAIN
      retry
    rescue EOFError
      break
    end
  end

  connection.close
end
```

启动之前用过的那个客户端，一直保持连接打开：

```
$ tail -f /var/log/system.log | nc localhost 4481
```

即便没有向服务器发送数据，read_nonblock调用仍然会立即返回。事实上，它产生了一个Errno::EAGAIN异常。下面是手册页中关于EAGAIN的描述：

> 文件被标记用于非阻塞式IO，无数据可读。

原来是这么回事。这不同于readpartial，后者在这种情况下会阻塞。

如果你碰上了这种错误该怎么办？套接字是否会阻塞？在本例中，我

们进入一个忙循环并不停地重试（retry）。不过这仅出于演示目的，而并非正确的做法。

对被阻塞的读操作进行重试的正确做法是使用IO.select：

```
begin
  connection.read_nonblock(4096)
rescue Errno::EAGAIN
  IO.select([connection])
  retry
end
```

上面的代码可以实现的功能同之前那种堆砌了大量read_nonblock与retry的代码一样，但却去掉了不必要的循环。使用套接字数组作为IO.select调用的第一个参数将会造成阻塞，直到其中的某个套接字变得可读为止。所以应该仅当套接字有数据可读时才调用retry。在下一章我们会更细致地讲解IO.select。

在本例中，我们使用非阻塞方法重新实现了阻塞式的read方法。这本身并没有什么用处。但是IO.select提供了一种灵活性，可以在进行其他工作的同时监控多个套接字或是定期检查它们的可读性。

什么时候读操作会阻塞？

read_nonblock方法首先检查Ruby的内部缓冲区中是否还有未处理的数据。如果有，则立即返回。

然后，read_nonblock会询问内核是否有其他可用的数据可供select(2)读取。如果答案是肯定的，不管这些数据是在内核缓冲区还是网络中，它们都会被读取并返回。其他情况都会使read(2)阻塞并在read_nonblock中引发异常。

11.2 非阻塞式写操作

非阻塞式写操作同我们之前看到的write调用有多处重要的不同。最明显的一处是：write_nonblock可能会返回部分写入的结果，而write调用总是将你发送给它的数据全部写入。

下面使用netcat启动一个临时服务器来演示这种行为：

```
$ nc -l localhost 4481
```

然后再启动一个采用了write_nonblock的客户端：

```
                              # ./code/snippets/write_nonblock.rb
require 'socket'

client = TCPSocket.new('localhost', 4481)
payload = 'Lorem ipsum' * 10_000

written = client.write_nonblock(payload)
written < payload.size  #=> true
```

当运行这两个程序时，从客户端处打印出了true。也就是说它返回了一个整数值，这个值小于负载（payload）数据的长度。write_nonblock方法之所以返回，是因为碰上了某种使它出现阻塞的情况，因此也就没法进行写入，所以返回了整数值，告诉我们写入了多少数据。接下来我们要负责将还未发送的数据继续写入。

write_nonblock的行为和系统调用write(2)一模一样。它尽可能多地写入数据并返回写入的数量。和Ruby的write方法不同的是，后者可能会多次调用write(2)写入所有请求的数据。

如果一次调用没法写入所有请求的数据，那该怎么办？显然应该试着继续写入没完成的部分。但别急着立刻下手。如果底层的write(2)仍处于阻塞，那你会得到一个`Errno::EAGAIN`异常。最终还是得靠`IO.select`，它可以告诉我们何时某个套接字可写，这意味着可以无阻塞地进行写入。

```
                    # ./code/snippets/retry_partial_write.rb
require 'socket'

client = TCPSocket.new('localhost', 4481)
payload = 'Lorem ipsum' * 10_000

begin
  loop do
    bytes = client.write_nonblock(payload)

    break if bytes >= payload.size
    payload.slice!(0, bytes)
    IO.select(nil, [client])
  end

rescue Errno::EAGAIN
  IO.select(nil, [client])
  retry
end
```

这里我们将一个套接字数组作为`IO.select`的第二个参数，这样`IO.select`会一直阻塞，直到其中的某个套接字可以写入。

例子中的循环语句正确地处理了部分写操作。当`write_nonblock`返回一个小于负载长度的整数时，我们将这部分数据从负载中截去，然后等套接字再次可写时，重新执行循环。

什么时候写操作会阻塞？

底层的write(2)在下述两种情况下会阻塞。

(1) TCP连接的接收端还没有确认接收到对方的数据，而发送方已经发送了所允许发送的数据量。TCP使用拥塞控制算法确保网络不会被分组所淹没。如果数据花费了很长时间才到达TCP连接的接收端，那么要注意不要发送超出网络处理能力的数据，以免网络过载。①

(2) TCP连接的接收端无力处理更多的数据。即便是另一端已经确认接收到了数据，它仍必须清空自己的数据窗口，以便重新填入其他数据。这就涉及内核的读缓冲区。如果接收端没有处理它接收的数据，那么拥塞控制算法会强制发送端阻塞，直到客户端可以接收更多的数据为止。

11.3　非拥塞式接收

尽管非阻塞式的read和write用得最多，但除了它们之外，别的方法也有非阻塞形式。

accept_nonblock和普通的accept几乎一样。还记不记得我当时说过accept只是从侦听队列中弹出一个连接？如果侦听队列为空，那accept就得阻塞了。而accept_nonblock就不会阻塞，只是产生一个Errno::EAGAIN。

① 如果发送时间过长，可能是因为网络出现了拥塞，那么这时就应该减少数据发送量，避免加剧拥塞。——译者注

下面是一个例子：

```
                              # ./code/snippets/accept_nonblock.rb
require 'socket'

server = TCPServer.new(4481)

loop do
  begin
    connection = server.accept_nonblock
  rescue Errno::EAGAIN
    # 执行其他重要的工作。
    retry
  end
end
```

11.4　非拥塞式连接

看看你现在能不能猜出connect_nonblock方法的用途？结果可能会有点出乎你的意料！connect_nonblock的行为和其他的非阻塞式IO方法有些不同。

其他方法要么是完成操作，要么是产生一个对应的异常，而conncet_nonblock则是保持操作继续运行，并产生一个异常。

如果connect_nonblock不能立即发起到远程主机的连接，它会在后台继续执行操作并产生Errno::EINPROGRESS，提醒我们操作仍在进行中。下一章会讲解如何得知后台操作已经完成。现在来看一个简短的例子：

```
                        # ./code/snippets/connect_nonblock.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')

begin
  # 在端口80向google.com发起一个非阻塞式连接。
  socket.connect_nonblock(remote_addr)
rescue Errno::EINPROGRESS
  # 操作在进行中。
rescue Errno::EALREADY
  # 之前的非阻塞式连接已经在进行当中。
rescue Errno::ECONNREFUSED
  # 远程主机拒绝连接。
end
```

第12章

连接复用

连接复用是指同时处理多个活动套接字。这并不是指并行处理，也和多线程无关。后面会用例子解释这一点。

根据目前所学到的技术，想象一下应该怎么编写一个需要随时处理多条TCP连接中的可用数据的服务器。我们可能会利用刚学到的有关非阻塞式IO的知识来避免在特定的套接字上陷入停滞。

```
                        # ./code/snippets/naive_multiplexing.rb
# 创建一个连接数组。
connections = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

# 进入无限循环。
loop do
  # 处理每个连接……
  connections.each do |conn|
    begin
      # 采用非阻塞的方式从每个连接中进行读取，
      # 处理接收到的任何数据，不然就尝试下一个连接。
      data = conn.read_nonblock(4096)
      process(data)
    rescue Errno::EAGAIN
    end
  end
end
```

这行得通吗？当然！不过需要频繁地执行循环。

每一次调用read_nonblock都要使用至少一个系统调用，如果没有数据可读，服务器会浪费大量的处理周期。如前所述，read_nonblock使用select(2)检查是否有可用数据。而有一个Ruby包装器，可以让我们按照自己的意图直接使用select(2)。

12.1 select(2)

下面是处理多个TCP连接中可用数据的更好的方法：

```
                          # ./code/snippets/sane_multiplexing.rb
# 创建一个连接数组。
connections = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

loop do
  # 查询select(2)哪一个连接可以进行读取了。
  ready = IO.select(connections)

  # 从可用连接中进行读取。
  readable_connections = ready[0]
  readable_connections.each do |conn|
    data = conn.readpartial(4096)
    process(data)
  end
end
```

这个例子使用IO.select极大地降低了处理多个连接的开销。IO.select的作用是接受若干个IO对象，然后告知哪一个可以进行读写，这样你就不必再像刚才那样瞎子摸象了。

来看一看IO.select的一些属性。

它可以告诉你文件描述符何时可以读写。在上面的例子中我们只给IO.select传递了一个参数，但实际上IO.select可以使用3个数组作为参数。

```
                              # ./code/snippets/select_args.rb
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

IO.select(for_reading, for_writing, for_writing)
```

第一个参数是希望从中进行读取的IO对象数组。第二个参数是希望进行写入的IO对象数组。第三个参数是在异常条件下使用的IO对象数组。大多数应用程序可以忽略第三个参数，除非你对带外数据（out-of-band data）感兴趣。（第18章会详述这方面的内容）。要注意的是，就算你只打算从单个IO对象中读取，你也得把它放到一个数组中传递给IO.select。

它返回一个数组的数组。IO.select返回一个包含了3个元素的嵌套数组，分别对应它的参数列表。第一个元素包含了可以进行无拥塞读取的IO对象。注意，这是IO.select首个参数的IO对象数组的一个子集。第二个元素包含了可以进行无拥塞写入的IO对象，第三个元素包含了适用异常条件的对象。

```
                           # ./code/snippets/select_returns.rb
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

ready = IO.select(for_reading, for_writing, for_writing)

# 对于每个作为参数传入的数组均会返回一个数组。
# 在这里，for_writing中没有连接可写，for_reading中有一个连接可读。
p ready #=> [[<TCPSocket>], [], []]
```

它会阻塞。IO.select是一个同步方法调用。按照目前的方法来使用它会造成阻塞，直到传入的某个IO对象状态发生变化。这时它会立刻返回。如果多个对象状态发生变化，那么全部都通过嵌套数组返回。

IO.select还有第四个参数：一个以秒为单位的超时值。它可以避免IO.select永久地阻塞下去。传入一个整数或浮点值指定超时。如果在IO状态发生变化之前就已经超时，那么IO.select会返回nil。

```
                              # ./code/snippets/select_timeout.rb
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

timeout = 10
ready = IO.select(for_reading, for_writing, for_writing,
    timeout)

# 在这里IO.select在10秒钟内没有检测到任何状态的改变，
# 因此返回nil，而非嵌套数组。
p ready #=> nil
```

> 你也可以传递纯Ruby对象给IO.select，只要它们适用于to_io并能返回一个IO对象。这样也有好处，如此一来你就无需维护IO对象到领域对象的映射了。如果实现了to_io方法，IO.select就可以使用纯Ruby对象了。

12.2　读/写之外的事件

我们已经学习了如何使用IO.select监视套接字的读写状态，实际上它也可以用于其他一些地方。

12.2.1 EOF

如果你在监视某个套接字的可读性时，它接收到了一个EOF，那么该套接字会作为可读套接字数组的一部分被返回。在对其进行读取时，取决于当时所使用的read(2)的版本，可能会得到一个EOFError或nil。

12.2.2 accept

如果你在监视某个服务器套接字的可读性时，它收到了一个接入的连接，那么该套接字会作为可读套接字数组的一部分返回。显然你需要对这种套接字进行特殊处理，应该使用accept而非read。

12.2.3 connect

这大概算得上是最有意思的部分了。上一章我们学习了connect_nonblock，了解到如果它不能立刻完成连接，则会产生Errno::EINPROGRESS。我们可以使用IO.select了解后台连接是否已经完成：

```
                      # ./code/snippets/multiplexing_connect.rb
require 'socket'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')

begin
  #发起到google.com端口80的非阻塞式连接。
  socket.connect_nonblock(remote_addr)
rescue Errno::EINPROGRESS
  IO.select(nil, [socket])

  begin
```

```
      socket.connect_nonblock(remote_addr)
    rescue Errno::EISCONN
      # 成功!
    rescue Errno::ECONNREFUSED
      # 被远程主机拒绝。
    end
  end
```

这段代码的第一部分和上一章的一模一样。尝试进行connect_nonblock并处理Errno::EINPROGRESS，这意味着连接过程发生在后台。接着看新代码部分。

我们利用IO.select监视套接字状态是否变得可写。如果发生改变，就可以确定底层的连接已经完成。为了获知状态，我们只需要再次试用connect_nonblock即可！如果它产生了Errno::EISCONN，就表明套接字已经连接到了远程主机。搞定！其他异常表明连接远程主机时出现了错误。

这段特别的代码实际上模拟了阻塞式的connect。为什么要这样？部分原因是为了展示可以用它做些什么， 你也可以像这样编写自己的代码。你可以发起connect_nonblock，撒手去做其他工作，随后调用带超时参数的IO.select。如果底层的连接还没有完成，你可以继续做别的事，随后再检查IO.select。

实际上我们可以利用这个小技巧使用Ruby编写一个非常简单的端口扫描器[①]。端口扫描器试图连接远程主机某个端口区域内的所有端口，并告诉你哪些端口是开放的，可以连接。

① http://en.wikipedia.org/wiki/Port_scanner.

```ruby
                                               # ./code/port_scanner.rb
require 'socket'

# 设置参数
PORT_RANGE = 1..128
HOST = 'archive.org'
TIME_TO_WAIT = 5 # 秒

# 为每个端口创建一个套接字并发起非阻塞式连接。
sockets = PORT_RANGE.map do |port|
  socket = Socket.new(:INET, :STREAM)
  remote_addr = Socket. sockaddr_in(port,'archive.org')

  begin
    socket.connect_nonblock(remote_addr)
  rescue Errno::EINPROGRESS
  end

  socket
end

# 设置期限。
expiration = Time.now + TIME_TO_WAIT

loop do
  # 我们每次调用IO.select并调整超时值，这样永远都不会超期。
  _, writable, _ = IO.select(nil, sockets, nil, expiration -
      Time.now)
  break unless writable

  writable.each do |socket|
    begin
      socket.connect_nonblock(socket.remote_address)
    rescue Errno::EISCONN
      # 如果套接字已经连接，那么我们就将此视为一次成功。
      puts "#{HOST}:#{socket.remote_address.ip_port} accepts
          connections..."
      # 将套接字从列表中移去，这样就不会再被选作可写的套接字。
```

```
        sockets.delete(socket)
      rescue Errno::EINVAL
        sockets.delete(socket)
      end
    end
  end
```

这段代码利用connect_nonblock一次性发起了几百个连接，然后使用IO.select对它们进行监视并确认哪一个连接能够顺利完成。下面是我在archive.org上运行的输出结果：

```
archive.org:25 accepts connections...
archive.org:22 accepts connections...
archive.org:80 accepts connections...
archive.org:443 accepts connections...
```

注意，结果未必是有序的。先完成处理的连接被先打印出来。这是一组很常见的开放端口，端口25保留用于SMTP，端口22保留用于SSH，端口80保留用于HTTP，端口443保留用于HTTPS。

12.3 高性能复用

IO.select来自Ruby的核心代码库。它是在Ruby中进行复用的唯一手段。大多数现代操作系统支持多种复用方法。select(2)几乎总是这些方法中最古老，也是用得最少的那个。

IO.select在少数情况下表现还不错，但是其性能同它所监视的连接数呈线性关系。监视的连接数越多，性能就越差。而且select(2)系统调用受到FD_SETSIZE的限制，它是一个定义在你本地C代码库中的宏。select(2)无法对编号大于FD_SETSIZE（在多数系统上这个数字是

1024）的文件描述符进行监视。因此`IO.select`最多只能监视1024个`IO`对象。

当然了，条条大路通罗马。

poll(2)系统调用与select(2)略有不同，不过这点不同也仅限于表面而已。Linux的epoll(2)以及BSD的kqueue(2)系统调用比select(2)和poll(2)效果更好、功能更先进。像EvenMachine这种高性能联网工具在可能的情况下更倾向于使用epoll(2)或kqueue(2)。

我不打算给出这些特定的系统调用的例子，我向你推荐一个叫做`nio4r`的Ruby gem[①]，它为所有这些复用方法提供了一个通用的接口，便于使用系统中性能最好的可用方法。

① https://github.com/tarcieri/nio4r.

第13章

Nagle算法

Nagle算法是一种默认应用于所有的TCP连接的优化。

这种优化最适合那些不进行缓冲、每次只发送很小数据量的应用程序。因而该算法通常被不满足这些条件的服务器禁用。接下来看一下这个算法。

程序向套接字执行写操作之后，有下面3种可能的结果。

(1) 如果本地缓冲区中有足够的数据可以组成一个完整的TCP分组[①]，那么就立即发送。

(2) 如果本地缓冲区中没有尚未处理的数据，接收端的数据也都全部已经确认接收，那么就立即发送。

(3) 如果接收端还有未确定的答复（acknowledgement），也没有足够的数据组成一个完整的TCP分组，那么就将数据放入本地缓冲区。

该算法避免了发送大量的微型TCP分组问题。它最初是设计用来解决

① 也称作full-size segment。它的具体大小由接收方的MSS（Maximum Segment Size）决定。——译者注

像telnet这种协议所存在的问题：在telnet中，伴随着每次击键，字符就会立刻发送到网络上。

如果你使用的是HTTP协议，它的请求/响应至少够组成一个TCP分组，因此Nagle算法除了会延缓最后一个分组发送之外，一般不会造成什么影响。该算法旨在避免在某些极特定的情况下搬石头砸自己的脚，比如在实现telnet时。考虑到Ruby的缓冲以及在TCP之上所实现的大部分常见的协议，你可能希望禁用Nagle算法。

例如，每个Ruby Web服务器都禁用了该选项。禁用方法如下：

```
                             # ./code/snippets/disable_nagle.rb
require 'socket'

server = TCPServer.new(4481)

# 禁用Nagle算法。告诉服务器不带延迟，即时发送。
server.setsockopt(Socket::IPPROTO_TCP, Socket::TCP_NODELAY, 1)
```

第14章

消息划分

有件事我们到现在都没有谈起，那就是如何将服务器与客户端之间交换的消息进行格式化。

CloudHash存在的一个问题是，客户端必须为发送到服务器的每条命令打开一个新连接。主要的原因是客户端/服务器并没有一致的方式来划分消息的起止位置，所以只能退而求次，使用EOF来表明消息的终止。

尽管这样也算把问题搞定了，但是并不理想。为每条命令打开一个新连接增加了不必要的开销。你可以在同一个TCP连接上发送多条消息，但如果你不打算关闭连接，那就需要有某种方式来表明消息之间的起止。

> 多条消息重用连接的想法同我们所熟悉的HTTP keep-alive特性背后的理念是一样的。在多个请求间保持连接开放（包括客户端和服务器协商的划分消息的方法），通过避免打开新的连接来节省资源。

说白了，有无数种方法可以用于在消息间进行划分。一些很复杂，一些很简单。取决于你想如何格式化消息。

协议与消息

我一直在谈论消息，它和协议并不是一回事。比如说，HTTP协议既定义了消息边界（连续的新行）也定义了用于消息内容（涉及请求行、头部等）的协议。

协议定义了应该如何格式化消息，不过本章关注的是如何在TCP流中分隔不同的消息。

14.1　使用新行

使用新行（newlines）的确是一种划分消息的简单方法。如果你确定应用程序的客户端和服务器都会运行在同类操作系统上，那你甚至可以在套接字上退而使用`IO#gets`和`IO#puts`发送带新行的消息。

让我们来重写`CloudHash`服务器的相关部分，使用新行（而非EOF）划分消息。

```
def handle(connection)
  loop do
    request = connection.gets
    break if request == 'exit'
    connection.puts process(request)
  end
end
```

对服务器的相关改动无非就是增加了`loop`，把`read`改成了`gets`。这样服务器就能够处理客户端的所有请求，直到其发送`'exit'`请求为止。

更健壮的方法是使用`IO.select`，在连接上等待事件发生。如果客户端套接字没有发送`'exit'`请求就断开连接，我们现阶段的服务器就会崩溃。

而客户端接下来会用以下方式送出请求：

```
def initialize(host, port)
  @connection = TCPSocket.new(host, port)
end

def get
  request "GET #{key}"
end

def set
  request "SET #{key} #{value}"
end

def request(string)
  @connection.puts(string)

  # 读取数据，直到收到新行获取答复。
  @connection.gets
end
```

注意，客户端不再使用类方法。现在连接能够保持多个请求，我们可以将一个连接封装成一个对象的实例，接下来调用该对象的方法即可。

新行与操作系统

前面曾讲述过，如果你确定客户端/服务器都运行在同类操作系统上，那么就可以使用gets和puts。我来解释一下为什么。

要是你查看gets和puts的文档，上面说到它使用$/作为默认的行分隔符。该变量中的值在Unix系统中是\n，而在Windows系统中则是\r\n。所以如果在一个系统中使用puts，在另一个系统中使用gets，可能会造成不兼容。如果你想使用这个方法，那么要确保给这些方法传递一个明确的行分隔符作为参数，这样就没有兼容性的问题了。

在现实中使用新行划分消息的协议是HTTP。下面是一个简短的HTTP请求：

```
GET /index.html HTTP/1.1\r\n
Host: www.example.com\r\n
\r\n
```

例子中使用转义序列\r\n明确地标出了新行。任何HTTP客户端/服务器都能够识别这些新行，不管它们运行在何种操作系统上。

这个方法肯定能解决问题，但并非唯一的解决方法。

14.2　使用内容长度

另一种划分消息的方法是指定内容长度（content length）。

利用这种方法，消息发送方先计算出消息的长度，使用pack将其转换成固定宽度的整数，后面跟上消息主体一并发送。消息的接收方首先读取这个长度值。这样就能知道消息的大小。然后接收方严格读取长度值所指定的字节数，获取完整的消息。

下面使用这个方法来修改ClouldHash的相关部分：

```
# 获得一个随机的固定宽度整数的大小。
SIZE_OF_INT = [11].pack('i').size

def handle(connection)
  # 用pack将消息长度转换成固定宽度的整数。对它进行read及unpack操作。
  packed_msg_length = connection.read(SIZE_OF_INT)
  msg_length = packed_msg_length.unpack('i').first

  # 读入给定长度的消息。
```

```
    request = connection.read(msg_length)
    connection.write process(request)
  end
```

客户端可以像下面这样发送请求：

```
payload = 'SET prez obama'

# 用pack将消息长度转换成固定宽度的整数。
msg_length = payload.size
packed_msg_length = [msg_length].pack('i')

# 写入消息长度以及消息主体。
connection.write(packed_msg_length)
connection.write(payload)
```

客户端用pack将消息长度转换成一个与处理器要求相一致的整数（native endian integer）。这一点非常重要，因为它保证了任何给定的整数都会被转化成同样数量的字节。如果不是这样，服务器就不知道究竟应该读取多少数位来得到消息的长度。用了这个方法，客户端/服务器在通信时总是会使用同样数量的字节来表示消息长度。

这次涉及的代码量稍微有点大，但是该方法没有使用任何像`gets`或`puts`这样的包装方法，只是使用了`read`和`write`这类基本IO操作。

第15章

超　　时

超时其实就是忍耐。你愿意在套接字连接上等待多长时间呢？套接字读取呢？套接字写入呢？

所有这些的答案都视你的忍耐力而定。高性能网络程序通常都不愿意等待那些没完没了的操作。如果你的套接字没能在5秒内完成数据写入，那就说明存在问题，应该采取其他的操作。

15.1　不可用的选项

如果你花时间研读过Ruby代码，可能看到过`timeout`库（标准库的一部分）。尽管这个库在Ruby中通常用于套接字编程，但我不打算在这里讨论它，因为有更好的超时处理方式。`timeout`库提供了一种通用的超时机制，但是操作系统也有一套针对套接字的超时机制，效果更好且更直观。

操作系统也提供了自带的套接字超时处理机制，可以通过套接字选项`SNDTIMEO`和`RCVTIMEO`进行设置。不过自Ruby 1.9起，这个特性就不能再用了。由于Ruby在有线程存在时对于阻塞式IO所采用的处理方

法，它将poll(2)相关的所有套接字操作进行了包装，这样操作系统自带的套接字超时就没有什么优势了。所以这些这类超时处理方式也没法用了。

那还剩下什么能用呢?

15.2 IO.select

啊，还是召唤我们的“老朋友”IO.select吧。它的用途可真够多的。

之前我们已经知道如何使用IO.select了。下面是将其用于超时的方法：

```
                              # ./code/snippets/read_timeout.rb
require 'socket'
require 'timeout'

timeout = 5 # 秒

Socket.tcp_server_loop(4481) do |connection|

  begin
    # 发起一个初始化read(2)。这一点很重要，
    # 因为要求套接字上有被请求的数据，有数据可读时避免使用select(2)。
    connection.read_nonblock(4096)

  rescue Errno::EAGAIN
    # 监视连接是否可读
    if IO.select([connection], nil, nil, timeout)
      # IO.select会将套接字返回，不过我们并不关心返回值，
      # 不返回nil就意味着套接字可读。
      retry

    else
```

```
      raise Timeout::Error
    end
  end

  connection.close
end
```

我在这里使用timeout只是为了访问Timeout::Error常量。

15.3 接受超时

就像我们之前看到的那样，accept能同IO.select很好的合作。如果你需要对accept设置超时，做法同read一样。

```
server = TCPServer.new(4481)
timeout = 5 # 秒

begin
  server.accept_nonblock
rescue Errno::EAGAIN
  if IO.select([server], nil, nil, timeout)
    retry
  else
    raise Timeout::Error
  end
end
```

15.4 连接超时

对连接设置超时的做法同其他示例差不多。

```
                        # ./code/snippets/connect_timeout.rb
require 'socket'
```

```
require 'timeout'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
timeout = 5 # 秒

begin
  # 发起到google.com端口80的非阻塞式连接。
  socket.connect_nonblock(remote_addr)

rescue Errno::EINPROGRESS
  # 表明连接过程正在进行中。
  # 我们监视套接字何时可读，这意味着连接已经完成。
  # 一旦再次进入上面的代码块，就会转入EISCONN rescue代码块，
  # 然后运行至begin代码块外，在这里套接字就已经可以使用了。
  if IO.select(nil, [socket], nil, timeout)
    retry
  else
    raise Timeout::Error
  end

rescue Errno::EISCONN
  # 表明连接已经顺利完成。
end

socket.write("ohai")
socket.close
```

这些基于超时的IO.select机制使用广泛，甚至在Ruby的标准库中也能看到，它们比操作系统自带的套接字超时处理机制的稳定性更高。

第16章

DNS查询

超时可以让你很好地控制代码，但是有些地方就没法那么随心所欲了。

看一下这个客户端连接的例子：

```
                           # ./code/snippets/client_easy_way.rb
require 'socket'

socket = TCPSocket.new('google.com', 80)
```

我们知道Ruby在构造函数内部调用了`connect`。因为我们传入的是主机名，而非IP地址，Ruby需要查询DNS将主机名解析成可以连接的IP地址。

害群之马？一个缓慢的DNS服务器会阻塞住整个Ruby进程。这可是多线程环境最糟糕的问题了。

MRI和GIL

标准Ruby实现（MRI[①]）包含了一个全局解释器锁（Global Interpreter

① Matz's Ruby Interpreter. ——译者注

Lock，GIL）。它确保Ruby解释器一次只做一件有潜在危险的事。在多线程环境中，它才能真正发挥作用。当一个线程进行活动时，其他线程全部处于阻塞状态。这使得MRI可以使用更安全、更简单的代码来编写。

好在GIL能够理解阻塞式IO。如果有一个线程在进行阻塞式IO（例如一个阻塞式`read`），MRI会释放GIL并让另一个线程继续执行。当阻塞式IO调用完成后，线程就等待下一次运行。

如果涉及C语言扩展，MRI可就不那么大方了。只要代码块用到了C语言扩展API，GIL就会阻塞其他代码的运行。阻塞式IO在这种情况下也不例外，如果一个C语言扩展阻塞在IO操作上，所有其他线程都会被阻塞。

目前问题的关键在于Ruby使用了一个C语言扩展查询DNS，所以如果DNS查询长时间阻塞，MRI就不会释放GIL。

resolv

所幸的是Ruby在标准中提供了该问题的解决方法。`resolv`库为DNS查询提供了一套纯Ruby的替代方案。这使得MRI能够为长期阻塞的DNS查询释放GIL。在多线程环境中这可谓是一大优势。

`resolv`库有自己的API，不过标准库中也包含了一个库，它对`Socket`进行了"猴子修补[①]"（monkey patch），使它也可以使用`resolv`：

```
require 'resolv' # 库
require 'resolv-replace' # 猴子修补
```

① http://en.wikipedia.org/wiki/Monkey_patch. ——译者注

第17章

SSL套接字

SSL使用公钥加密提供了一套用于在套接字上进行安全的数据交换的机制。

SSL套接字并没有取代TCP套接字，而是将不安全的套接字“升级”到安全的SSL套接字。如果你愿意，可以在TCP套接字之上再增添一个安全层。

要注意的是，一个套接字可以升级成SSL，但是一个套接字不能同时进行SSL和非SSL通信。当使用SSL时，端到端的通信必须全部都使用SSL完成，否则无法保证安全。

对于那些既需要在SSL上又需要在不安全的TCP上运行的服务，需要使用两个端口（两个套接字）。HTTP就是一个常见的例子：不安全的HTTP默认使用端口80，而HTTPS（运行在SSL之上的HTTP）通信默认是在端口443上。

所有TCP套接字都可以转换成SSL套接字。在Ruby中这通常使用标准库中的`openssl`实现。下面是一个例子：

```
                              # ./code/snippets/ssl_server.rb
require 'socket'
require 'openssl'

def main
  # 建立TCP服务器。
  server = TCPServer.new(4481)

  # 建立SSL环境。
  ctx = OpenSSL::SSL::SSLContext.new
  ctx.cert, ctx.key = create_self_signed_cert(
    1024,
    [['CN', 'localhost']],
    "Generated by Ruby/OpenSSL"
  )
  ctx.verify_mode = OpenSSL::SSL::VERIFY_PEER

  # 建立TCP服务器的SSL包装器。
  ssl_server = OpenSSL::SSL::SSLServer.new(server, ctx)

  # 在SSL套接字上接受连接。
  connection = ssl_server.accept

  # 处理连接。
  connection.write("Bah now")
  connection.close
end

# 接下来的代码直接取自webrick/ssl，
# 它生成一个适用于Context对象使用的自签名SSL证书。
def create_self_signed_cert(bits, cn, comment)
  rsa = OpenSSL::PKey::RSA.new(bits){|p, n|
    case p
    when 0; $stderr.putc "." # BN_generate_prime
    when 1; $stderr.putc "+" # BN_generate_prime
    when 2; $stderr.putc "*" # 搜索good prime,
      # n = #of try,
      # 数据同样取自BN_generate_prime。
```

```
    when 3; $stderr.putc "\n" # 找到good prime, n==0 - p, n==1 - q,
      # 数据同样取自BN_generate_prime
    else; $stderr.putc "*" # BN_generate_prime
    end
  }
  cert = OpenSSL::X509::Certificate.new
  cert.version = 2
  cert.serial = 1
  name = OpenSSL::X509::Name.new(cn)
  cert.subject = name
  cert.issuer = name
  cert.not_before = Time.now
  cert.not_after = Time.now + (365*24*60*60)
  cert.public_key = rsa.public_key

  ef = OpenSSL::X509::ExtensionFactory.new(nil,cert)
  ef.issuer_certificate = cert
  cert.extensions = [
    ef.create_extension("basicConstraints","CA:FALSE"),
    ef.create_extension("keyUsage", "keyEncipherment"),
    ef.create_extension("subjectKeyIdentifier", "hash"),
    ef.create_extension("extendedKeyUsage", "serverAuth"),
    ef.create_extension("nsComment", comment),
  ]
  aki = ef.create_extension("authorityKeyIdentifier",
                            "keyid:always,issuer:always")
  cert.add_extension(aki)
  cert.sign(rsa, OpenSSL::Digest::SHA1.new)

  return [ cert, rsa ]
end

main
```

这段代码生成一个自签名的SSL证书，用它来支持SSL连接。这个证书是SSL安全性的关键所在。没有了它，连接的安全性基本上只是镜花水月。

同样，出于安全性考虑，还应该设置verify_mode = OpenSSL::SSL::VERIFY_PEER。很多Ruby代码库默认将该值设置为OpenSSL::SSL::VERIFY_NONE。这是一个比较宽松的设置，它允许未经验证的SSL证书，放弃了很多由证书提供的安全性。该问题在Ruby社区有过详尽的讨论。①

一旦服务器开始运行，你可以试着用一个普通的TCP连接，配合netcat进行连接：

```
$ echo hello | nc localhost 4481
```

然后服务器就会崩溃并产生OpenSSL::SSL::SSLError。这可是件好事！

服务器拒绝接受来自一个不安全的客户端的连接，然后产生一个异常。重启服务器，使用配备了SSL的Ruby客户端再次连接：

```
                              # ./code/snippets/ssl_client.rb
require 'socket'
require 'openssl'

# 创建TCP客户端套接字。
socket = TCPSocket.new('0.0.0.0', 4481)

ssl_socket = OpenSSL::SSL::SSLSocket.new(socket)
ssl_socket.connect

ssl_socket.read
```

① http://www.rubyinside.com/how-to-cure-nethttps-risky-default-https-behavior-4010.html.

现在你应该可以看到服务器和客户端都顺利退出了。服务器与客户端之间完成了一次成功的SSL通信。

在典型的产品设置中，不会让你去生成自签名证书（这只适合开发/测试）。你通常要从一个可信任机构购买证书。该机构会提供给你用于安全通信的cert和key，要用它们代替上面例子中的自签名证书。

第18章

紧急数据

之前我强调过TCP套接字提供了一种有序的数据流。换句话说，你可以将TCP数据流想象成一个队列。套接字连接的一端向连接中写入数据，就相当于将数据入列。这些数据经过若干个阶段（本地缓冲、网络传输、远程缓冲），然后在接收端的套接字处出列。

这种思考模型对于典型的TCP通信同样适用。TCP紧急数据，更多的时候被称作“带外数据”（out-of-band data），支持将数据推到队列的前端，绕过其他已在传送途中的数据，以便于连接的另一端尽快接收到这些数据。

Socket还有一个我们没有讲过的方法Socket#send。

Socket#send像一个特殊的Socket#write（继承自IO）。实际上，如果不带参数，它的行为和write一样。

```
# 两者的效果相同。
socket.write 'foo'
socket.send 'foo'
```

write方法可以由所有的IO对象使用，而send方法则由套接字专用。

这使得Socket#send可以接受第二个参数：标志（flag）。我们可以为send指定一个标志，表明某些数据为紧急数据。

18.1　发送紧急数据

接下来看看以下代码：

```
                         # ./code/snippets/sending_urgent_data.rb
require 'socket'

socket = TCPSocket.new 'localhost', 4481

# 使用标准方法发送数据。
socket.write 'first'
socket.write 'second'

# 发送紧急数据。
socket.send '!', Socket::MSG_OOB
```

要发送1B的紧急数据，我们需要调用send并将Socket::MSG_OOB常量作为标志。这里的OOB指的就是带外数据。

这就是发送紧急数据的方法，不过这还不足以使接收方优先获取到紧急数据。换句话说，发送方和接收方需要合作才能完成这项工作。

18.2　接受紧急数据

下面是接收方使用Socket#recv获取紧急数据的方法：

```
                       # ./code/snippets/receiving_urgent_data.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
```

```
    # 优先接收紧急数据。
    urgent_data = connection.recv(1, Socket::MSG_OOB)

    data = connection.readpartial(1024)
  end
```

要接收紧急数据，需要使用Socket#recv以及在发送紧急数据时用过的那个标志。同Socket#send一样，Socket#revc专由套接字用于数据读取。它同样可以接受标志作为参数。

即便是“普通”数据先于紧急数据写入，也可以在“普通”数据之前使用紧急数据，而这正是我们希望达到的效果。要注意的是，必须明确地接受紧急数据。如果不调用recv，服务器就不会注意到紧急数据。也就是说，如果接收方没有去查找紧急数据，那么它什么都得不到。继续往下看如何处理这种情况。

> 如果不存在未处理的紧急数据，调用connection.recv(1, Socket::MSG_OOB)则会失败，并产生Errno::EINVAL。

18.3　局限

你大概已经注意到在上面的例子中我只发送了一个字节的紧急数据。这是有意而为的。TCP实现对于紧急数据仅提供了有限的支持，一次只能发送一个字节的紧急数据。如果你要发送多个字节，那么只有最后一个字节会被视为紧急数据。之前的那些数据会视为普通的TCP数据流。

18.4　紧急数据和 IO.select

和大多数情况下一样，我们可以使用IO.select监视多个套接字上的紧急数据。不过该方法需要特别留意。

还记不记得我说过IO.select的第三个参数是一个IO对象数据？我们需要获取这些IO对象上的带外数据。下面是具体的做法：

```
                                # ./code/snippets/select_args.rb
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

IO.select(for_reading, for_writing, for_reading)
```

有没有注意到这里传递了一个用于监视读取的套接字数组作为第三个参数？这意味着，如果这些套接字接收到了紧急数据，它们会被包含在IO.select所返回数组的第三个元素中。

这挺不错，这样我们就可以监视套接字上的紧急数据，而不用再盲目的调用recv了。不过根据我的经验，IO.select会不停地报告有紧急数据，即便是所有的紧急数据都已经处理完毕！这种情况会持续到处理完一些普通的TCP数据流，很可能直到本地接收缓冲区为空时才能停止。这意味着你得增加一些额外的错误处理或是状态跟踪，以确保不会陷入处理并不存在的紧急数据的循环。

鉴于这方面的问题以及单个字节的限制，紧急数据是一个很少用到的TCP特性。在Ruby标准库中，它只用到了一次（在net/ftp中），而且还错误地试图发送不止一个字节的紧急数据。[1]

① http://www.ruby-forum.com/topic/201519.

18.5　SO_OOBINLINE 选项

另一种处理紧急数据的方法是将其放入普通数据流。有一个叫做SO_OOBINLINE的套接字选项，允许在带内接收带外数据。也就是说，紧急数据会依照次序被合并到普通数据流中。启用该选项后，紧急数据就不会再受到优待。它依据写入次序从队列中读出。

下面是启用该选项的方法：

```
                               # ./code/snippets/oob_inline.rb
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # 在带内随同其他普通数据接收紧急数据。
  connection.setsockopt :SOCKET, :OOBINLINE, true

  # 注意，当遇到紧急数据时停止读取。
  connection.readpartial(1024) #=> 'foo'
  connection.readpartial(1024) #=> '!'
end
```

在本例中，数据foo在'!'之前被接收，所以紧急数据不再会被优先接收。我特意演示出read系列方法处理紧急数据的方式。尽管数据foo和'!'都处于1024B的读取限制中，当遇到紧急数据时，它会停止读取，返回获得的数据，然后再重新开始读取。

该选项只对接收方套接字有效，对发送方套接字无效。

第19章

网络架构模式

之前的章节涵盖了基础知识和必备技能，接下来的部分将转向最佳实践与真实案例。前面那些章类似参考文档：你可以向前翻阅，回顾某些特性的用法或是问题的解决之道，不过仅限于你清楚自己想要查找的具体内容。

如果你的任务是用Ruby编写一个FTP服务器，仅了解本书的前半部分肯定是有帮助的，但这些知识没法让你创造出伟大的软件。

尽管你了解建造模块，但是你还不知道构架网络应用程序的常见方式。如何处理并发？如何处理错误？处理缓慢的客户端的最好方法是什么？如何最有效地利用资源？

这类问题正是本书后面部分要解答的。接下来我们先学习6种网络架构模式，然后将它们应用到一个案例项目中。

实现思路

与其用一堆图表和抽象的描述，我更喜欢的说明问题的方式是，采用

一个能够实现的案例项目并用不同的架构重复实现它。这样才能真正理解不同架构之间的差异。

出于这个原因，我们要编写一个包含了部分FTP功能的服务器。为什么只包含了部分功能？因为我希望将注意力放在架构模式，而非协议实现上。那为什么选择FTP？因为这样就不用再编写单独的客户端就可以进行测试了。现成的FTP客户端已经够多了。

对于那些不熟悉FTP的读者，FTP代表文件传输协议（File Transfer Protocol）。它定义了一套基于文本的协议，通常运行在TCP之上，用于在两台计算机之间传送文件。

如你所见，这有点像是在浏览文件系统。FTP同时使用了两个TCP套接字。一个是控制套接字，用于在服务器和客户端之间发送FTP命令及参数。每当传送文件时，就会使用一个新的TCP套接字。这是个不错的技巧，它使得在传送文件的同时仍可以在控制套接字上处理命令。

下面是FTP服务器的协议实现。它定义了一些常用的方法，用于写入格式化的FTP响应以及建立控制套接字，它还提供了一个`CommandHandler`类，封装了基于每个连接的单独命令的处理。这一点很重要。同一服务器上的每个连接可能有不同的工作目录，`CommandHandler`类也考虑到了这一点。

```
                              # ./code/ftp/common_handler.rb
module FTP
  class CommandHandler
    CRLF = "\r\n"

    attr_reader :connection
```

```
def initialize(connection)
  @connection = connection
end

def pwd
  @pwd || Dir.pwd
end

def handle(data)
  cmd = data[0..3].strip.upcase
  options = data[4..-1].strip

  case cmd
  when 'USER'
    # 接受匿名用户
    "230 Logged in anonymously"

  when 'SYST'
    # 用户名?
    "215 UNIX Working With FTP"

  when 'CWD'
    if File.directory?(options)
      @pwd = options
      "250 directory changed to #{pwd}"
    else
      "550 directory not found"
    end

  when 'PWD'
    "257 \"#{pwd}\" is the current directory"

  when 'PORT'
    parts = options.split(',')
    ip_address = parts[0..3].join('.')
    port = Integer(parts[4]) * 256 + Integer(parts[5])

    @data_socket = TCPSocket.new(ip_address, port)
    "200 Active connection established (#{port})"
```

```
      when 'RETR'
        file = File.open(File.join(pwd, options), 'r')
        connection.respond "125 Data transfer starting
            #{file.size} bytes"

        bytes = IO.copy_stream(file, @data_socket)
        @data_socket.close

        "226 Closing data connection, sent #{bytes} bytes"

      when 'LIST'
        connection.respond "125 Opening data connection
            for file list"

        result = Dir.entries(pwd).join(CRLF)
        @data_socket.write(result)
        @data_socket.close

        "226 Closing data connection, sent #{result.size} bytes"

      when 'QUIT'
        "221 Ciao"
      else
        "502 Don't know how to respond to #{cmd}"
      end
    end
  end
end
```

这个协议的实现并没有过多关注网络和并发，这部分内容将在下一章介绍。

第 20 章

串行化

我们要学习的第一个网络架构模式是处理请求的串行化模型。我们接着之前 FTP 服务器继续讲解。

20.1 讲解

在串行化架构中，所有的客户端连接是依次进行处理的。因为不涉及并发，多个客户端不会同时接受服务。

串行化架构的处理流程很直观：

(1) 客户端连接；

(2) 客户端/服务器交换请求及响应；

(3) 客户端断开连接；

(4) 返回到步骤(1)。

20.2 实现

```
# ./code/ftp/arch/serial.rb
```

```
require 'socket'
require_relative '../command_handler'

module FTP
  CRLF = "\r\n"

  class Serial
    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
      trap(:INT) { exit }
    end

    def gets
      @client.gets(CRLF)
    end

    def respond(message)
      @client.write(message)
      @client.write(CRLF)
    end

    def run
      loop do
        @client = @control_socket.accept
        respond "220 OHAI"

        handler = CommandHandler.new(self)

        loop do
          request = gets

          if request
            respond handler.handle(request)
          else
            @client.close
            break
          end
        end
```

```
      end
    end
  end
end

server = FTP::Serial.new(4481)
server.run
```

注意，`Serial` 类只负责联网和并发操作，协议处理部分交由 `CommandHandle` 的方法来处理。接下来你会经常看到这种模式。让我们来从头看起。

```
                                # ./code/ftp/arch/serial.rb
class Serial
  def initialize(port = 21)
    @control_socket = TCPServer.new(port)
    trap(:INT) { exit }
  end

  def gets
    @client.gets(CRLF)
  end

  def respond(message)
    @client.write(message)
    @client.write(CRLF)
  end
```

这 3 个方法属于这类特定实现的样板代码（boilerplate）。`initialize` 方法打开一个套接字，由该套接字接受客户端连接。

`gets` 方法将 `gets` 委托给当前客户端连接。注意，它传递了一个明确的分隔符，用以保证在具有不同默认分隔符的平台之间的可移植性。

respond 方法用来写入格式化过的 FTP 响应。message 中包含了整数类型的响应代码以及对应的字符串消息。当客户端接收到回车符\r 和换行符\n 组合时，它就知道已经获得了完整的响应信息。

```
                                      # ./code/ftp/arch/serial.rb
  def run
    loop do
      @client = @control_socket.accept
      respond "220 OHAI"

      handler = CommandHandler.new(self)
```

这是服务器的主循环，所有的处理逻辑都发生在外部主循环之内。

循环中唯一一次调用 accept 就是你在这儿看到的这次。它接受一个来自 @control_socket 的连接，后者在 initialize 中进行初始化。代码为 220 的响应内容属于协议实现细节。FTP 要求在接受一个新的客户端连接之后要打声招呼[①]。

最后一处是为该连接进行 CommandHandler 的初始化。该类封装了服务器上每个连接的当前状态（当前工作目录）。我们可以将接入的请求交给 handler 对象，然后获得对应的响应。

这部分代码是串行化模式中进行并发操作的绊脚石。在这部分代码进行处理时，服务器没法继续接受新的连接，更不用说实现并发了。当我们学到其他模式如何应对这种情况时，就会明显看出它们之间的差异了。

① 回复代码 220（reply code 220)表示 Service ready for new user，详见 RFC959。
——译者注

```
                              # ./code/ftp/arch/serial.rb
loop do
  request = gets

  if request
    respond handler.handle(request)
  else
    @client.close
    break
  end
end
```

这部分完成了我们的 FTP 服务器的串行化实现。

在内部循环中，使用 `gets` 从客户端套接字中获取带有显式分隔符的请求。然后将获得的请求传给 `handler`，由它为客户端构造对应的响应信息。

鉴于这是一个功能完善的 FTP 服务器（尽管只支持部分 FTP 功能），我们实际上可以运行该服务器，使用标准的 FTP 客户端进行连接，看看它的实际表现：

```
$ ruby code/ftp/arch/serial.rb
```

```
$ ftp -a -v 127.0.0.1 4481
cd /var/log
pwd
get kernel.log
```

20.3 思考

很难明确地归纳每种模式的优劣，因为这完全取决于你的需求。我会

尽力解释每种模式最适用的场景及其所作出的一些权衡。

串行化架构最大的优势在于它的简单性。没有锁，没有共享状态，处理完一个连接之后才能处理另一个。在资源使用方面亦是如此：一个实例处理一个连接，一个萝卜一个坑，绝不多消耗资源。

串行化架构明显的劣势是不能并发操作。即便是当前连接处于空闲，也不能处理等待的连接。同样，如果某个连接使用的链路速度不佳，或者在发送请求之间暂停，那么服务器就只能保持阻塞，直到连接关闭。

对于随后那些更有意思的模式而言，串行化模式仅仅是一个起点而已。

第21章

单连接进程

这是首个可以对请求进行并行处理的网络架构。

21.1 讲解

要支持并发处理，只需要将串行化架构略加修改即可。接受连接的代码不需要改动，处理来自套接字数据的代码也保持不变。

相关改动出现在接受连接之后，服务器会fork出一个子进程，这个子进程的唯一目的就是处理新连接。连接处理完毕之后就退出。

进程衍生的基础知识

只要你使用$ruby myapp.rb启动程序，就会生成一个新的Ruby进程来载入并执行代码。

如果在程序中使用了fork，那实际上就是在运行期间创建了一个新进程。fork可以使你获得两个一模一样的进程。新创建的进程被视为“孩子”；原先的进程被视为“双亲”。一旦fork完成，就拥有了两个进程，它们可以各行其道，各行其事。

> 这一点及其有用，这意味着我们可以accept一个连接，fork一个子进程，这个子进程会自动获得一份客户端连接的副本。无需其他的设置、数据共享或者锁，直接就可以开始并行处理了。

让我们来理清楚事件流程：

(1) 一个连接抵达服务器；

(2) 主服务器进程接受该连接；

(3) 衍生出一个和服务器一模一样的新子进程；

(4) 服务器进程返回步骤1，由子进程并行处理连接。

得益于内核语义，这些进程是并行执行的。子进程处理连接时，原先的父进程可以继续接受新连接，衍生出新的子进程对其进行处理。

不管何时，总是有一个父进程等着接受连接，但可能会有多个子进程分别处理单个连接。

21.2 实现

```
                    # ./code/ftp/arch/process_per_connection.rb
require 'socket'
require_relative '../command_handler'

module FTP
  class ProcessPerConnection
    CRLF = "\r\n"

    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
```

```
      trap(:INT) { exit }
    end

    def gets
      @client.gets(CRLF)
    end

    def respond(message)
      @client.write(message)
      @client.write(CRLF)
    end

    def run
      loop do
        @client = @control_socket.accept

        pid = fork do
          respond "220 OHAI"

          handler = CommandHandler.new(self)

          loop do
            request = gets

            if request
              respond handler.handle(request)
            else
              @client.close
              break
            end
          end
        end

        Process.detach(pid)
      end
    end
  end
end
```

```
server = FTP::ProcessPerConnection.new(4481)
server.run
```

如你所见，大部分代码都没有变动。最大的不同在于内循环被放在了一个fork调用中。

```
                    # ./code/ftp/arch/process_per_connection.rb
@client = @control_socket.accept

pid = fork do
  respond "220 OHAI"

  handler = CommandHandler.new(self)
```

使用accept接受连接后，服务器进程立刻就使用代码块调用fork。新的子进程会对该代码块进行求值，然后退出。

这意味着每一个接入的连接都由一个独立的进程进行处理。父进程不会对代码块求值。它只会沿着自己的执行路径进行。

```
                     # /code/ftp/arch/process_per_connection.rb
Process.detach(pid)
```

注意，我们在最后调用了Process.detach。在一个进程退出之后，它并不会被完全清除，直到其父进程查询该进程的退出状态。在这里我们并不关心子进程的退出状态是什么，所以提前把它与父进程分离，确保子进程退出后，所占用的资源能够完全清除。[1]

① 如果你希望学习更多有关进程生成及僵尸进程的知识，可以参考我翻译的另一本书《理解Unix进程》（*Working with Unix Processes*）。

21.3 思考

该模式有诸多优势。首先是它的简单性。为了能够并行处理多个客户端，只需要在串行化实现的基础上增加极少量的代码即可。

第二个优势是这种并行操作不难理解。之前我说过，fork实际上提供了一个子进程所需要的所有东西的副本。不用留心边界情况（edge cases），没有锁或竞争条件，只是简单的分离而已。

一个明显的劣势是，对于fork出的子进程的数量没有施加上限。如果客户端数量不大，这倒也不是什么问题，但如果生成了上百个进程，那你的系统就可能会崩溃了。这方面可以使用第23章介绍的Preforking模式解决。

取决于操作环境，使用fork可能会有问题。只有Unix系统才支持fork。这就意味着在Windows或JRuby中就没法用fork了。

另一个方面是究竟该用进程还是线程。我把这个问题留到下一章来讨论，届时我们会接触到线程。

21.4 案例

- shotgun
- inetd

第22章

单连接线程

22.1 讲解

单连接线程模式和上一章的单连接进程模式非常相似。不同之处在于，它是生成线程，而非进程。

> **线程与进程**
>
> 线程和进程都可以用于并行操作，但是方式大不相同。没有万能药，究竟用哪个取决于实际情况。
>
> **生成**（spawn）。就生成而言，线程的生成成本要低得多。生成一个进程需要创建原始进程所拥有的一切资源的副本。线程以进程为单位，多个线程都存在于同一个进程中。由于多个线程共享内存，无需创建副本，因而线程的生成速度要快得多。
>
> **同步**。因为线程共享内存，当使用会被多个线程访问的数据结构时，一定要多加小心。这通常意味着要在线程之间使用互斥量（mutex）、锁和同步访问。进程就无须如此了，因为每个进程都有自己的一份资源副本。

并行。两者都提供了由内核实现的并行计算功能。关于MRI中的线程并行需要注意的一件重要的事情是：解释器对当前执行环境使用了一个**全局**锁。因为线程以进程为单位，这意味着它们都运行在同一个解释器中。即便是使用了多线程，MRI也使得它们无法实现真正的并行。在另外一些Ruby实现中，如JRuby或Rubinius 2.0，不存在这样的问题。

进程没有这方面的麻烦，因为每次生成新的进程，它都会获得自己的一份Ruby解释器的副本，所以也就无需全局锁。在MRI中，只有进程才能实现真正的并发。

关于并行和线程还要多说一点。即便是MRI使用了全局解释器锁，它对线程的处理也非常巧妙。第16章我提到过，如果某个线程阻塞在IO上，Ruby能让其他的线程继续执行。

总而言之，线程是轻量级的，进程是重量级的。两者都用于并行操作。两者都有各自的适用环境。

22.2 实现

```
                      # ./code/ftp/arch/thread_per_connection.rb
require 'socket'
require 'thread'
require_relative '../command_handler'

module FTP
  Connection = Struct.new(:client) do
    CRLF = "\r\n"

    def gets
      client.gets(CRLF)
    end
```

```
  def respond(message)
    client.write(message)
    client.write(CRLF)
  end

  def close
    client.close
  end
end

class ThreadPerConnection
  def initialize(port = 21)
    @control_socket = TCPServer.new(port)
    trap(:INT) { exit }
  end

  def run
    Thread.abort_on_exception = true

    loop do
      conn = Connection.new(@control_socket.accept)

    Thread.new do
      conn.respond "220 OHAI"

      handler = FTP::CommandHandler.new(conn)

      loop do
        request = @conn.gets

        if request
          conn.respond handler.handle(request)
        else
          conn.close
          break
        end
      end
```

```
          end
        end
      end
    end
  end

  server = FTP::ThreadPerConnection.new(4481)
  server.run
```

这段代码同之前的两个例子略微有些不同。使用的方法基本一样，但是组织形式却截然不同。

```
                      # ./code/ftp/arch/thread_per_connection.rb
Connection = Struct.new(:client) do
  CRLF = "\r\n"

  def gets
    client.gets(CRLF)
  end

  def respond(message)
    client.write(message)
    client.write(CRLF)
  end

  def close
    client.close
  end
end
```

此处的样板代码和之前看到的一样，但是如今被放入了Connection类中，而不是直接定义在服务器类中。

```
                      # ./code/ftp/arch/thread_per_connection.rb
def run
```

```
Thread.abort_on_exception = true

loop do
  conn = Connection.new(@control_socket.accept)

  Thread.new do
    conn.respond "220 OHAI"

    handler = FTP::CommandHandler.new(conn)
```

这里有两处关键的不同。首先是这部分代码生成了一个线程，而在上个例子的这个地方生成的是一个进程。其次是从accept返回的客户端套接字被传给了Connection.new；每个线程均获得自己的Connection实例。

使用线程时，这一点非常重要。如果我们就像以前那样，只是简单地将客户端套接字分配给一个实例变量，那么它会在所有的活动线程之间共享。因为这些线程是从一个共享的FTP服务器实例中生成的，所以它们会共享该实例的内部状态。

这与同进程打交道有着显著的差别，在后者中每个进程都会获得内存中所有资源的副本。之所以有些开发者声称线程编程不容易，其中一个原因便是状态共享。如果你使用线程进行套接字编程，有一条简单的经验：让每个线程获得它自己的连接对象。这可以让你少些麻烦。

22.3　思考

该模式和前一个模式有很多共同的优势：代码修改量很少，很容易理解。尽管使用线程会引入锁以及同步问题，但这里我们并不用担心这

个问题，因为每个连接是由单个独立线程来处理的。

该模式较单连接进程的一个优势是线程占用资源少，因而可以获得数量上的增加。比起使用进程，它能为客户端服务提供更好的并发性。

不过先等等，别忘了MRI GIL使得这一优势无法变成现实。归根结底，没有哪个模式能够所向披靡。每一种都应该思考、尝试、检验。

这个模式和单连接进程模式有一个共同的劣势：线程数会不断增加，直到系统不堪重负。如果你的服务器要处理持续增加的连接，系统可能难以在所有的活动线程上进行维护及切换。这可以通过限制活动线程数解决。在第24章我们会讲解这种做法。

22.4 案例

- WEBrick
- Mongrel

第23章

Preforking

23.1 讲解

Preforking模式又折回到之前的单连接进程架构上了。

它依赖进程作为并行操作的手段，但并不为每个接入的连接衍生对应的子进程，而是在服务器启动后，连接到达之前就先衍生出一批进程。

下面是处理流程：

(1) 主服务器进程创建一个侦听套接字；

(2) 主服务器进程衍生出一大批子进程；

(3) 每个子进程在共享套接字上接受连接，然后进行独立处理；

(4) 主服务器进程随时关注子进程。

这个流程的重点是，主服务器进程打开侦听套接字，却并不接受该套接字之上的连接。它然后衍生出预定义数量的一批子进程，每个子进程都有一份侦听套接字的副本。子进程在各自的侦听套接字上调用`accept`，不再考虑父进程。

这个模式的精妙之处在于，无须担心负载均衡或是子进程连接的同步，因为内核已经替我们完成这个工作了。对于多个进程试图在同一个套接字的不同副本上接受（accept）连接的问题，内核会均衡负载并确保只有一个套接字副本可以接受某个特定的连接。

23.2 实现

```
                              # ./code/ftp/arch/preforking.rb
require 'socket'
require_relative '../command_handler'

module FTP
  class Preforking
    CRLF = "\r\n"
    CONCURRENCY = 4

    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
      trap(:INT) { exit }
    end

    def gets
      @client.gets(CRLF)
    end

    def respond(message)
      @client.write(message)
      @client.write(CRLF)
    end

    def run
      child_pids = []

      CONCURRENCY.times do
```

```
      child_pids << spawn_child
    end

    trap(:INT) {
      child_pids.each do |cpid|
        begin
          Process.kill(:INT, cpid)
        rescue Errno::ESRCH
        end
      end

      exit
    }

    loop do
      pid = Process.wait
      $stderr.puts "Process #{pid} quit unexpectedly"

      child_pids.delete(pid)
      child_pids << spawn_child
    end
  end

  def spawn_child
    fork do
      loop do
        @client = @control_socket.accept
        respond "220 OHAI"

        handler = CommandHandler.new(self)

        loop do
          request = gets

          if request
            respond handler.handle(request)
          else
            @client.close
```

```
              break
            end
          end
        end
      end
    end
  end
end

server = FTP::Preforking.new(4481)
server.run
```

这个实现同我们迄今所看过的有明显不同。我们分两部分进行讨论，先从上半部分开始。

```
                              # ./code/ftp/arch/preforking.rb
def run
  child_pids = []

  CONCURRENCY.times do
    child_pids << spawn_child
  end

  trap(:INT) {
    child_pids.each do |cpid|
      begin
        Process.kill(:INT, cpid)
      rescue Errno::ESRCH
      end
    end

    exit
  }

  loop do
    pid = Process.wait
    $stderr.puts "Process #{pid} quit unexpectedly"
```

```
    child_pids.delete(pid)
    child_pids << spawn_child
  end
end
```

首先多次调用spawn_child，具体次数基于存储在CONCURRENCY中的值而定。spawn_child方法（随后会详细讲解）会fork一个新进程然后返回其进程id（pid），该值是唯一的。

生成子进程后，父进程为INT信号定义了一个信号处理器。当你键入Ctrl-C时，进程就会收到该信号。这个信号处理器仅用于将父进程接收到的INT信号转发给它的子进程。记住，子进程独立于父进程所存在，即便是父进程结束了，子进程也不会受到影响。所以对于父进程而言，在退出之前清理自己的子进程就显得很重要了。

信号处理完之后，父进程就进入了Process.wait循环。该方法会一直阻塞到有子进程退出为止。它返回退出子进程的pid。因为子进程并不应该退出，所以我们将这视为出现了异常情况。随后在STDERR上打印一条信息并生成一个新的子进程代替。

在一些Preforking服务器中，尤其是Unicorn①，父进程承担了更为活跃的角色，它还负责监视自己的子进程。例如，父进程可能会查看是否有哪个子进程耗费了太多的时间处理请求。如果是，父进程就会强行终止该子进程并生成新的子进程取代它。

① http://unicorn.bogomips.org.

```
                            # ./code/ftp/arch/preforking.rb
def spawn_child
  fork do
    loop do
      @client = @control_socket.accept
      respond "220 OHAI"

      handler = CommandHandler.new(self)

      loop do
        request = gets

        if request
          respond handler.handle(request)
        else
          @client.close
          break
        end
      end
    end
  end
end
```

这种方法的核心部分应该很熟悉了。这次它被放入了`fork`和`loop`中。因此新进程在调用`accept`之前就已经衍生出来了。最外层的循环确保每个连接处理并关闭后，继续处理新的连接。通过这种方法，每个子进程都处于它们各自的连接接受循环中。

23.3 思考

这个巧妙的模式得益于以下几处设计。

相较于类似的单连接进程架构，Preforking不用在每个连接期间进行

fork。进程衍生的成本可不少，在单连接进程架构中，每个连接都要承担由此带来的开销。

如前所述，因为该模式提前就生成了所有的进程，因而避免了进程过量的情况。

比起与Preforking类似的线程模式，这个模式的一个优势就是完全隔离。因为每个进程都拥有包括Ruby解释器在内的所有资源的副本，单个进程中的故障不会影响其他进程。因为线程共享进程资源以及内存空间，单线程故障可能会无法预测地影响到其他线程。

Preforking的一个劣势是：衍生的进程越多，消耗的内存也越多。进程可不是免费的午餐。考虑到每个衍生的进程都会获得所有资源的一份副本，可以预料到每一次进程衍生，内存占用率就要增加100%（以父进程为基准）。

按照这种衍生方式，占用100MB内存的进程在衍生出4个子进程之后将占用500MB内存。即便这样，也只有4个并发连接。

我不打算在这里就这一点再喋喋不休了，不过这种模式的代码的确简单。尽管需要理解几个概念，但总的来说并不难，也不用担心运行期间出问题。

23.4 案例

- Unicorn

第24章

线程池

24.1 讲解

线程池模式之于Preforking，一如单连接线程与单连接进程之间的关系。同Preforking差不多，线程池在服务器启动后会生成一批线程，将处理连接的任务交给独立的线程来完成。

这个架构的处理流程和前一个一样，只需要把“进程”改成“线程”就行了。

24.2 实现

```
                              # ./code/ftp/arch/thread_pool.rb
require 'socket'
require 'thread'
require_relative '../command_handler'

module FTP
  Connection = Struct.new(:client) do
    CRLF = "\r\n"
```

```
  def gets
    client.gets(CRLF)
  end

  def respond(message)
    client.write(message)
    client.write(CRLF)
  end

  def close
    client.close
  end
end

class ThreadPool
  CONCURRENCY = 25

  def initialize(port = 21)
    @control_socket = TCPServer.new(port)
    trap(:INT) { exit }
  end

  def run
    Thread.abort_on_exception = true
    threads = ThreadGroup.new

    CONCURRENCY.times do
      threads.add spawn_thread
    end

    sleep
  end

  def spawn_thread
    Thread.new do
```

```
          loop do
            conn = Connection.new(@control_socket.accept)
            conn.respond "220 OHAI"

            handler = CommandHandler.new(self)

            loop do
              request = conn.gets

              if request
                conn.respond handler.handle(request)
              else
                conn.close
                break
              end
            end
          end
        end
      end
    end
  end

  server = FTP::ThreadPool.new(4481)
  server.run
```

这里再次出现了两个方法。一个用来生成线程，另一个用来封装线程生成以及线程行为。因为这里用到了线程，所以还要使用Connection类。

```
                                # ./code/ftp/arch/thread_pool.rb
CONCURRENCY = 25

def initialize(port = 21)
  @control_socket = TCPServer.new(port)
  trap(:INT) { exit }
```

```
  end

  def run
    Thread.abort_on_exception = true
    threads = ThreadGroup.new

    CONCURRENCY.times do
      threads.add spawn_thread
    end

    sleep
  end
```

这个方法创建了一个ThreadGroup跟踪所有的线程。ThreadGroup有点像一个可对线程进行操作的数组（a thread-aware Array）。你可以向ThreadGroup中加入线程，当某个线程成员执行结束后，它就会从这个组中丢弃。

你可以使用ThreadGroup#list获得组中当前所有活动线程列表。在这个实现中，我们其实并没有用到这个技巧，但如果你想对所有的活动线程进行操作（例如使用join），那么ThreadGroup就能派上用场了。

同上一章大同小异，我们依据CONCURRENCY的值多次调用spanwn_thread。注意，这里CONCURRENCY的值要比Preforking中的高。这还是因为线程的开销更小，所以可以使用更多的线程。要记住的是MRI GIL减少了一部分由此带来的收益。

方法的最后调用了sleep避免退出。当线程池中的线程有工作任务时，主线程保持空闲。理论上来说它可以监视线程池，不过这里只是用了sleep不让其退出。

```
                           # ./code/ftp/arch/thread_pool.rb
def spawn_thread
  Thread.new do
    loop do
      conn = Connection.new(@control_socket.accept)
      conn.respond "220 OHAI"

      handler = CommandHandler.new(self)

      loop do
        request = conn.gets

        if request
          conn.respond handler.handle(request)
        else
          conn.close
          break
        end
      end
    end
  end
end
```

这个方法平淡无奇，没什么出彩之处。它和Preforking一样。生成一个线程，重复执行连接处理代码。同样由内核确保一个连接只能由单个线程接受。

24.3 思考

有关线程池模式大部分的思考内容同前一个模式一样。

除了那些线程与进程之间显而易见的权衡之外，线程池模式不需要每

次处理连接时都生成线程，也没有什么令人抓狂的锁或竞争条件，但却仍提供了并行处理能力。

24.4　案例

- Puma

第25章

事件驱动

迄今为止我们看到的这些模式其实都是串行化模式的变体而已。其他几种模式实际上使用的结构与串行化模式相同，只不过包装了线程或者进程。

事件驱动（Reactor）模式采用的是一种和之前完全不同的方法。

25.1　讲解

事件驱动模式（基于Reactor模式[1]）如今可谓风头正劲。它也是EventMachine、Twisted，Node.js以及Nginx等库的核心所在。

该模式结合了单线程和单进程，它至少可以达到之前模式所能提供的并行操作级别。

它以一个中央连接复用器（被称为Reactor核心）为中心。连接生命周期中的每个阶段都被分解成单个的事件，这些事件之间可以按照任意的次序交错并处理。连接的不同阶段只是可能的IO操作而已：

① http://en.wikipedia.org/wiki/Reactor_pattern.

accept、read，write以及close。

中央复用器监视所有活动连接的事件，在触发事件时分派相关的代码。

下面是事件驱动模式的工作流程：

(1) 服务器监视侦听套接字，等待接入的连接；

(2) 将接入的新连接加入到套接字列表进行监视；

(3) 服务器现在要监视活动连接以及侦听套接字；

(4) 当某个活动连接可读时，服务器从该连接读取一块数据并分派相关的回调函数；

(5) 当某个活动连接仍然可读时，服务器读取另一块数据并再次分派回调函数；

(6) 服务器收到另一个新连接，将其加入套接字列表进行监视；

(7) 服务器注意到第一个连接已经可以写入，因而将响应信息写入该连接。

记住，所有的一切都发生在单个线程中。注意，第一个连接仍在读/写过程时，服务器就可以accept新连接了。

服务器将每次操作分割成小块，这样属于多个连接的不同事件就可以彼此交错了。

接下来研究实现代码。

25.2 实现

```ruby
                                          # ./code/ftp/arch/evented.rb
require 'socket'
require_relative '../command_handler'

module FTP
  class Evented
    CHUNK_SIZE= 1024 * 16

    class Connection
      CRLF = "\r\n"
      attr_reader :client

      def initialize(io)
        @client = io
        @request, @response = "", ""
        @handler = CommandHandler.new(self)

        respond "220 OHAI"
        on_writable
      end

      def on_data(data)
        @request << data

        if @request.end_with?(CRLF)
          # 完成请求。
          response @handler.handle(@request)
          @request = ""
        end
     end

     def respond(message)
       @response << message + CRLF
```

```
      #立即加载可以写入的任何内容，
      #其余部分将在下次套接字可写入时重试。
      on_writable
    end

    def on_writable
      bytes = client.write_nonblock(@response)
      @response.slice!(0, bytes)
    end

    def monitor_for_reading?
      true
    end

    def monitor_for_writing?
      !(@response.empty?)
    end
  end

  def initialize(port = 21)
    @control_socket = TCPServer.new(port)
    trap(:INT) { exit }
  end

  def run
    @handles = {}

    loop do
      to_read = @handles.values.select(&:monitor_for_
          reading?).map(&:client)
      to_write = @handles.values.select(&:monitor_for_
          writing?).map(&:client)

      readables, writables = IO.select(to_read + [@control_
          socket], to_write)

      readables.each do |socket|
        if socket == @control_socket
```

```
            io = @control_socket.accept
            connection= Connection.new(io)
            @handles[io.fileno] = connection

          else
            connection= @handles[socket.fileno]

            begin
              data = socket.read_nonblock(CHUNK_SIZE)
              connection.on_data(data)
            rescue Errno::EAGAIN
            rescue EOFError
              @handles.delete(socket.fileno)
            end
          end
        end

        writables.each do |socket|
          connection= @handles[socket.fileno]
          connection.on_writable
        end
      end
    end
  end
end

server = FTP::Evented.new(4481)
server.run
```

这个实现采用了一种同之前那些实现不同的手法。我们把代码分解成几个部分研究。

```
                                    # ./code/ftp/arch/evented.rb
class Connection
```

这行代码定义了一个Connection类作为事件驱动服务器。

在前面几个线程模式的示例中，我们用Connection类保持进程间的状态隔离。 这个示例没有用线程，为什么需要Connection类呢?

所有基于进程的模式都使用进程隔离连接。不管利用进程的方法如何，它们总是确保无论何时都由单个独立的进程处理单个连接，所以每个连接基本上都是由一个进程描述。

事件驱动模式用的是单线程，但可以同时处理多个用户连接，所以它需要使用一个对象来描述每个独立的连接，这样就不会破坏连接各自的状态。

```
                                  # ./code/ftp/arch/evented.rb
class Connection
  CRLF = "\r\n"
  attr_reader :client

  def initialize(io)
    @client = io
    @request, @response = "", ""
    @handler = CommandHandler.new(self)

    respond "220 OHAI"
    on_writable
```

Connection类的开始部分看起来有些眼熟。

Connection类将底层的IO对象存储在它的@client实例变量中，外界可以通过attr_accessor对其进行访问。

当连接初始化完毕后，它会像从前一样获得自己的CommandHandler实例。随后它写入FTP所要求的定制的'hello'响应。不过并非直接写入客户端连接，而是将响应的主体信息写入@response变量。下面我

们会看到这将引发Reactor接管操作并将数据发送到客户端。

```
                                    # ./code/ftp/arch/evented.rb
def on_data(data)
  @request << data

  if @request.end_with?(CRLF)
    # 完成请求
    respond @handler.handle(@request)
    @request = ""
  end
end

def respond(message)
  @response << message + CRLF

  # 立即加载可以写入的任何内容
  # 其余部分将在下次套接字可写入时重试
  on_writable
end

def on_writable
  bytes= client.write_nonblock(@response)
  @response.slice!(0, bytes)
```

Connection的这部分定义了若干与Reactor核心进行交互的生命周期方法。

例如，当Reactor从客户端连接读取数据时，它触发on_data处理数据。在这个方法的内部，检查接收到的是否是一个完整的请求。如果是，会请求 @handler 建立对应的响应并将其赋给 @response。

当客户端连接可以进行写入时就调用on_writable方法。这就要和 @response 变量打交道了。它将 @response 中的内容写入客户端连

接。根据能够写入的字节数，将成功写入的数据从 @response 中移除。

这样，随后的写操作只会写入 @response 中本次没能写入的部分内容。如果能够写入全部内容，那么 @response 就变成了一个空字符串，就无法再进行写操作了。

monitor_for_reading?和monitor_for_writing?这两个方法被Reactor用来查询是否应该监视特定连接的读写状态。在本例中，只要有新的数据，我们都希望进行读取。如果 @response 有内容可写，我们希望获知可以进行写入的时机。如果 @response 中没有内容，即便是客户端连接可以写入，Reactor也不会发出通知。

```
                                          # ./code/ftp/arch/evented.rb
    def monitor_for_writing?
      !(@response.empty?)
    end
  end

  def initialize(port = 21)
    @control_socket = TCPServer.new(port)
    trap(:INT) { exit }
```

这就是Reactor核心的主要工作。

@handles 看起来像这样：{6 => #<FTP::Evented::Connection:xyz123>}，其中键对应的是文件描述符编号，值对应的是Connection对象。

主循环的第一行查询每一个活动连接，看是否需要使用之前介绍的生命周期方法对其进行读/写监视。对于有需要的连接，它获取其低层IO

对象的引用。

Reactor 随后将这些 IO 实例传给不带超时参数的 `IO.select`。`IO.select` 会一直阻塞到某个受监控的套接字出现值得关注的事件为止。

注意，Reactor 还会监视 `@control_socket` 是否可读，以便检测到新接入的客户端连接。

```
                                        # ./code/ftp/arch/evented.rb
def run
  @handles = {}

  loop do
    to_read = @handles.values.select(&:monitor_for_reading?).
        map(&:client)
    to_write = @handles.values.select(&:monitor_for_
        writing?).map(&:client)

    readables, writables = IO.select(to_read + [@control_
        socket], to_write)

    readables.each do |socket|
      if socket == @control_socket
        io = @control_socket.accept
        connection= Connection.new(io)
        @handles[io.fileno] = connection

      else
        connection= @handles[socket.fileno]

        begin
          data = socket.read_nonblock(CHUNK_SIZE)
          connection.on_data(data)
        rescue Errno::EAGAIN
        rescue EOFError
```

Reactor的这部分代码根据它从IO.select中接收到的事件触发对应的方法。

它首先处理可读的套接字。如果 @control_socket 可读，就意味着出现了一个新的客户端连接。Reactor调用accept接受连接，建立一个新的Connection并将其放入 @handles 散列中，这样就可以在下一轮循环中进行监视了。

接下来要处理可读的套接字是普通的客户端连接的情况。在这种情况下，代码会尝试读取数据，触发对应Connection的on_data方法。如果读操作出现阻塞（Errno::EAGAIN），不做特殊处理，让事件落空即可。如果客户端断开连接（EOFError），那么要确保从 @handles 散列中删除相应的条目，使得对应的对象可以被回收并不再受到监视。

代码最后一部分通过触发对应Connection的on_writable方法处理可写的套接字。

25.3　思考

事件驱动模式同其他模式有着显著的不同，因而也就产生了尤为不同的优势和劣势。

首先，该模式以极高的并发处理能力而闻名，能够处理成千上万的并发连接。光这一点就让其他模式无法望其项背，因为它们都受到进程/线程数量的限制。

如果服务器需要生成5000个线程来处理5000个连接，服务器估计会不

堪重负。就处理并发连接而言，事件驱动模式可谓一枝独秀并广为流传。

它主要的劣势是所施加的编程模型。一方面这个模型更简单，因为无需处理众多进程/线程。这意味着就不存在共享内存、同步、越界进程，等等。但是考虑到所有的并发都发生在单个线程内部，有一条非常重要的规则必须遵循，那就是绝不能阻塞Reactor。

要诠释这一点，让我们来仔细查看一下实现代码。在`CommandHandler`类中，当处理FTP文件传输命令（`RETR`）时，它实际上打开了一个套接字，以流的方式发送数据，然后关闭套接字。重要的是这个套接字是在Reactor主循环之外使用的，Reactor对其一无所知。

假设客户端在一条速度缓慢的连接上请求文件传输。这会对Reactor造成怎样的影响？

考虑到一切都运行在同一个线程之内，单个迟缓的客户端连接会阻塞住整个Reactor！当Reactor在`Connecton`上触发某个方法时，Reactor会一直阻塞到该方法返回为止。由于`on_data`方法委派（delegate）给了`CommandHandler`，当它以数据流的方式向客户端进行文件传输时，Reactor一直处于阻塞。在这期间，无法读取其他数据，也无法接受新的连接。

应用程序需要达成的任何事情都应该快速完成，这一点非常重要。那我们应该怎样使用Reactor处理缓慢的连接呢？利用Reactor自身！

如果你采用该模式，那就需要确保所有阻塞式IO都由Reactor自己来处理。在这个例子中就意味着由`CommandHandler`所使用的套接字需要被封装到`Connection`的子类中，它定义了自己的一套`on_data`和

on_writable方法。

当Reactor可以向缓慢的连接中写入数据时，它会触发相应的on_writable方法，该方法能够在没有阻塞的情况下尽可能多的向客户端写入数据。这样Reactor就可以在等待这个缓慢的远程连接的同时继续处理其他连接，一旦那条远程连接再次可用，仍可对其进行处理。

简而言之，事件驱动模式提供了一些显而易见的优势，真正简化了套接字编程的某些方面。另一方面，它需要你重新考虑自己的应用程序中所涉及的全部IO操作。该模式所带来的益处很容易就会被一些迟钝的代码或者含有阻塞式IO的第三方代码库搞得烟消云散。

25.4　案例

- EventMachine
- Celluloid::IO
- Twisted

第26章

混合模式

这是本书有关网络模式的最后一部分。这部分并不涉及特定的模式本身，而是阐述一个混合模式的概念，它采用了若干个之前学习过的模式。

尽管这些体系都可以应用到任何类型的服务中（在之前的章节中用的是FTP），但如今大量的注意力都投向了HTTP服务器。鉴于Web的广泛流行，这倒是也毫不意外。Ruby社区冲锋在Web运动的前线，在各类HTTP服务器中占有相当的份额。因此我们在这一章中看到的真实案例全都是HTTP服务器。

让我们来看一些例子吧。

26.1　nginx

nginx项目[①]提供了一个用C语言编写的性能极高的网络服务器，项目网站上宣称它能够在单服务器上服务100万个并发请求。nginx在Ruby

① http://nginx.org.

世界中多作为Web应用服务器前端的HTTP代理，不过它也能够处理HTTP，SMTP等协议。

nginx是如何实现如此高的并发的呢？

在核心部分[①]，nginx使用了Preforking模式。但是在每个衍生进程中使用的却是事件驱动模式。作为一个高性能的网络服务器，这种选择意义重大，原因如下。

首先，在nginx衍生子进程时，所有的相关成本在启动时就已经付出了。这就保证了ngnix能够最大限度地利用多核以及服务器资源。其次，事件驱动模式也贡献了一臂之力，它不进行任何生成（spawn），也不使用线程。使用线程的一个问题就是需要由内核承担所有活动进程的管理以及上下文切换所带来的开销。

ngnix疾如闪电的运行速度少不了其他一干特性的协助，这包括的严密的内存管理（只能利用C语言实现），但在核心部分，它混合使用了前面几章介绍的模式。

26.2 Puma

Puma rubygem提供了一个“专注于并发的Ruby Web服务器”。[②]Puma被设计作为由Ruby实现的王牌HTTP服务器，由于它大量倚重线程，所以并没有使用GIL（Rubinius或JRuby）。Puma的自述文件[③]对于其

① http://www.aosabook.org/en/nginx.html.
② http://puma.io.
③ https://github.com/puma/puma#description.

适用场景给出了很好的概述，并提及了GIL对于线程化的影响。

那么Puma是如何实现并发的呢?

在高层上，Puma利用线程池提供并发。主线程一直用于`accept`新的连接，然后将连接加入线程池待作处理。这便是不适用keep-alive[①]的HTTP连接的处理方法。不过Puma也支持HTTP的keep-alive。在处理连接时，如果首个请求要求连接保持活跃状态，那么Puma会尊重这一请求，不关闭连接。

不过这时Puma就不再只是`accept`这个连接了。它需要监视该连接上的新请求并进行处理。这是通过事件驱动类型的reactor实现的。当新的请求出现在处于保持活跃状态的连接上时，该请求会被再次加入线程池进行处理。

Puma的请求处理总是由线程池完成的。它通过一个能够监视所有持久连接的Reactor实现。

Puma同样包括了其他方面的精心优化，但其核心同样也是采用了多个前面几章介绍的模式。

26.3 EventMachine

EventMachine是Ruby圈里知名的事件驱动IO库。它利用Reactor模式提供高稳定性及可扩展性。EventMachine是用C语言编写的，但是通过C扩展的形式提供了Ruby接口。

---

① http://en.wikipedia.org/wiki/HTTP_persistent_connection.

那么EventMachine是如何实现并发的呢？

EventMachine的核心是利用事件驱动模式实现的。它是一个单线程的事件循环，可以处理多个并发连接上的网络事件。EventMachine也提供了一个线程池，用于推迟处理会拖累Reactor的那些耗时或阻塞式的操作。

EventMachine支持包括监视生成进程、网络协议实现等大量特性。利用多种架构只是它实现并发性的一种手段。

第27章

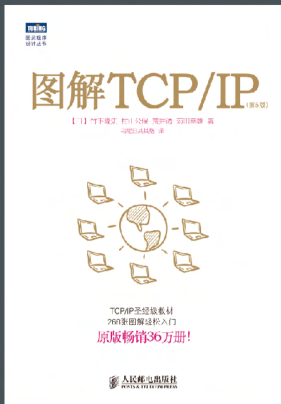
结　　语

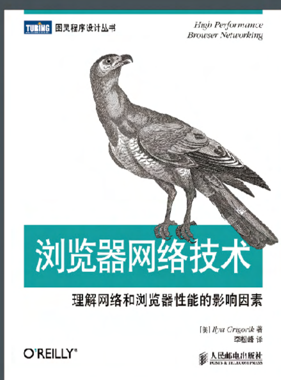
我确信你现在已经掌握了有关套接字编程基础及其扩展内容。你可以将它们应用于Ruby以及其他编程领域。这些都是非常有用的知识。

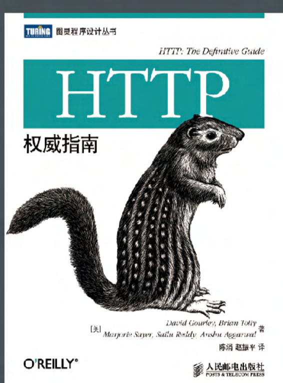
感谢你阅读本书。希望它能帮你更深入地理解工作中用到的技术。我的电子邮件是jesse@jstorimer.com，我非常乐意与你就本书或是任何相关的编程话题进行交流。

索　　引

重点推荐


图解TCP/IP
TCP/IP圣经级教材
260张图解轻松入门
原版畅销36万册!
人民邮电出版社


图灵程序设计丛书
High Performance Browser Networking
浏览器网络技术
理解网络和浏览器性能的影响因素
[美] Ilya Grigorik 著
李松峰 译
O'REILLY
人民邮电出版社


图灵程序设计丛书
HTTP: The Definitive Guide
HTTP
权威指南
[美] David Gourley, Brian Totty, Marjorie Sayer, Sailu Reddy, Anshu Aggarwal 著
陈涓 赵振平 译
O'REILLY
人民邮电出版社

读者评论

“我用Ruby做网络编程好几年了，但仍从这本书中学到很多新知识。”

“使用TCP Sockets的程序员和对它感兴趣的人最好都读读这本书。”

你知道Web服务器如何打开套接字并绑定到地址以及如何接受连接吗？作者在深入理解网络协议栈的工作机制之前，就做过大量Web编程。所以放下手中那本1000多页的网络手册吧！本书旨在为开发人员介绍套接字编程的方方面面。读完这本书后，你就会理解套接字编程的必备知识，能够编写服务器/客户端库以及并发网络程序。

本书中所有的代码均使用Ruby编写，但书中所讲述的内容并不仅限于Ruby。Berkeley套接字API有超过25年的应用历史，与所有的现代编程语言有着紧密的联系 。当使用Python、Go、C或其他编程语言时，这里所学的知识同样适用。本书介绍的都是网络编程的必备知识，你必然可以从中受益良多。

主要内容包括：

- 服务器和客户端的生命周期；
- 使用Ruby在合适的时机，以各种方式读取并写入数据；
- 提高套接字性能的一些方法；
- SSL套接字基础知识；
- 实现并发网络的6种架构模式；
- 连接复用、非阻塞IO、套接字超时和套接字选项，等等。

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/网络编程

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-33052-9

9 787115 330529 >

ISBN 978-7-115-33052-9
定价:29.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn